Revised 5/84

Assembly Language Programming

Objective Of This Module

This module offers an introduction to assembly language programming and the Atari Assembler Editor. Through the activities in this module you will see how assembly language is a particularly good language for fast, smooth animation.. You also will find that assembly language requires programming in great detail. Hopefully, you will find the rewards of a successful assembly language program are well worth the hard work.

Overview

1. The Assembler.
   What is the assembler and what does it do?

2. Assembly Language Format.
   What is the correct syntax and punctuation for assembly language programs?

3. 6502 Assembly Language Instruction Set.
   This section offers you an opportunity to experiment with various assembly language instructions.

4. Indexed Addressing Modes.
   The eight different addressing modes available on the Atari are explained and demonstrated.

5. Animation.
   In this section you will write an assembly language program that moves a spinning pinwheel around on the screen with a joystick.

6. The USR Function.
   This section explains how to call an assembly language routine from a BASIC program.

Prerequisite Concepts

1. You must have completed the Machine Architecture Module before doing this module.

Materials Needed

1. An Atari Assembler Editor Cartrigde and the User Manual.
2. An Advanced Topics Diskette.

The Assembler

This section explains how assembly language programs are executed and the assembler editor's role in the process.

In the Machine Architecture Module you recently completed, you had a chance to see some assembly language instructions and learn how the 6502 executes a program. You also learned that, regardless of what language you are programming in, the 6502 only understands machine language. How then does assembly language get converted to machine language in order for the CPU to execute your program?

Writing and executing assembly language programs requires an "assembler editor." You have already used the Atari Assembler Editor cartridge to execute assembly language programs in the Machine Architecture Module. When you insert your assembler cartridge in the Atari and turn on the computer, two programs on a chip inside the cartridge are available. One of the programs, called the "assembler," is responsible for converting your assembly language program to machine language. The second program, called the "editor," enables the programmer to type and edit the assembly language program before it is "assembled" to machine language by the assembler.

The assembly language program that a programmer writes and types into the computer is called the "source code." The programmer uses the editor to insert, delete, or alter any part of the source code. The source code includes the three-letter assembly language instructions, variable names, memory addresses, and labels. Listed below is the source code for a program that prints an arrow in the upper left hand corner of the screen. The program simply loads the accumulator with the Internal Character Set code number for an arrow, $7D. ($7D is the hexadecimal equivalent to 125 in base ten.) The $7D is then stored in screen RAM in order to print the arrow on the screen.

```
*=$0600      ;ORIGIN OF PROGRAM
LDA #$7D     ;LOAD ACCUMLATOR WITH CODE FOR AN ARROW
STA $9C40    ;SCREEN RAM LOCATION
BRK          ;DISCONTINUE PROGRAM EXECUTION
```

If you look at the right hand side of the program, you will notice that the source code includes remarks and explanations about what the program does. These comments are comparable to REM statements in BASIC. In assembly language you use a ";" to indicate that a remark follows, the same way you use a REM in BASIC. However, comments in assembly language are much more vital than in BASIC because of the difficulty people have understanding assembly language code.

Before this assembly language program can be executed, it must be passed through the assembler. The assembler reads through the source code and converts the program to machine language, a numerical code which the microprocesssor can understand. The assembler ignores the comments because they are not pertinent information to the CPU. The comments are only useful to the person who is trying to understand the program. The machine language version of the program is called the "object code." If you look to the left of the source code in the diagram below, you will see the object code. Note that the object code is listed in hexadecimal.

```
Object Code              Source Code

0000            0100     *=$0600          ;ORIGIN OF PROGRAM
0600 A97D       0110     LDA #$7D         ;LOAD ACC. WITH ARROW
0602 8D409C     0120     STA $9C40        ;SCREEN RAM LOCATION
0605 00         0130     BRK              ;DISCONTINUE PROGRAM
```

As the assembler converts the source code to object code, it stores the hexadecimal values in successive memory locations. The first instruction of the program, *=$0600, instructs the assembler to store the object code in memory starting at $600. The column on the far left of the object code above holds the addresses of where the object code is stored in memory. The numbers just to the right of the memory addresses comprise the object code, which has been stored in memory. For a closer look at how the object code has been stored in memory, see the diagram below.

```
Object Code in     Source Code
   Memory

$600  | A9 |      LDA #$7D      ;LOAD THE ACC. WITH ARROW
$601  | 7D |
$602  | 8D |      STA $9C40     ;STORE ACC. IN SCREEN RAM
$603  | 40 |
$604  | 9C |
$605  | 00 |      BRK           ;DISCONTINUE PROGRAM
```

A code number called the "opcode" has been stored in memory for each instruction. For example, A9 is the opcode for the LDA instruction. The CPU recognizes the A9 as a "load the accumulator" instruction. The opcodes are called opcodes because they are the "code" numbers that tell the microprocessor what "operation" to perform. The 8D (STA) in memory location $602 instructs the CPU to store the value in the accumulator into the specified location. All opcodes are one byte in length, so they take up one memory location.

The number following an instruction in the source code is called the "operand." It is called the operand because it is the number the CPU will be "operating on" when it executes the instruction. For example, the $7D following the LDA is the number the CPU will load into the accumulator. This will be explained in more depth in the next section. However, note that the operand is stored in memory directly after the opcode for the instruction. Also note that the entire object code is listed in hexadecimal numbers.

To summarize, the assembler converts the source code, or English-like version of the program, to object code. The object code is the machine language version of the program, which the assembler stores in memory. The object code is the specific set of instructions that the microprocessor will execute. The object code is made up of opcodes, which are the instructions to the CPU, and operands, which are the data to be operated on. Turn to Assembly Language Programming Worksheet #1 to take a closer look at some source code and object code.

Assembly Language Programming Worksheet #1

You will need an Assembler Editor Cartridge and an Advanced Topics Diskette to complete this worksheet.

1. Boot up the system with the Assembler Editor Cartridge and the Advanced Topics Diskette. You should have the EDIT prompt in the upper left hand corner of your screen. Load the ARROW program from the Advanced Topics Diskette into memory.

Type: `ENTER #D:ARROW`

2. Now type LIST and press <RETURN>. What type of code

do you see, source code or object code? ________________

3. To execute the program, the source code must be converted to object code by the assembler.

Type: ASM and press <RETURN>

The combined source code and object code should scroll up on the screen. The code you see on the screen should be the same as the code listed below.

```
0000          0100      *=   $600      ;ORIGIN OF PROGRAM
0600 A97D     0110      LDA #$7D       ;LOAD ACC. WITH ARROW
0602 8D409C   0120      STA $9C40      ;SCREEN RAM LOCATION
0605 00       0130      BRK            ;DISCONTINUE PROGRAM
```

4. We know that the opcode for LDA is A9 and the opcode for STA is 8D. What is the opcode for BRK?_____

5. Now run the program.

Type: BUG and press <RETURN>

You should see the word BUG on the screen. The Atari Assembler Editor executes the program from the "debugger." The debugger is another program on the assembler cartridge; it enables you to look at or change the contents of specific memory locations. Don't worry if you don't understand this. However, if you would like to learn more about how to use the debugger, see chapter 5, "Using the Debugger," of the Assembler Editor User's Manual.

6. Now you must clear the screen. Press the <SHIFT> and <CLEAR> keys at the same time. If you executed the program with an instruction at the bottom of the screen, once the program had been executed, the screen would scroll up and arrow will no longer be visible.

Type: <SHIFT><CLEAR>

7. To execute the program from the debugger, you have to tell the computer where the object code is stored in memory. The program is stored at memory location $600.

Type: G600   and press <RETURN>

The "G" stands for GO. Use the GO command to instruct the debugger to execute the program followed by the starting address of the program.

8. Try changing the character printed on the screen to another character by completing the steps below. First, you must return to the editor.

Type: X and press <RETURN>

To see the source code again,

Type: LIST and press <RETURN>

By holding down the <CTRL> key while pressing one of the arrow keys, you can move your cursor up to edit your source code. Place the cursor over the 7 in the #$7D, following the LDA instruction. Type in another number and press <RETURN>. Then go back to the debugger to execute the program by typing BUG.

Type: BUG and press <RETURN>

To clear the screen,

Type: <SHIFT><CLEAR>

Run the program to see what character you stored in screen memory. To execute your new program,

Type: G600 and press <RETURN>

The values for the internal character set are used to store letters in screen RAM to be displayed on the screen. The internal character set values are listed in a chart at the back of this module. Try experimenting with putting specific letters on the screen. The values are listed in decimal, so you must convert them to hexadecimal to use them in this program.

9. To see how fast the CPU is putting the arrow on the screen, run a program called ARW2 on the Advanced Topics Diskette. ENTER the ARW2 program into memory.

Type: ENTER ARW2 and press <RETURN>

The ARW2 program loads the accumulator with the value for an arrow, and then stores it in screen RAM, just as the ARROW program did. However, the ARW2 program stores a zero in screen RAM where the arrow was placed to show how fast the arrow is displayed and then erased. Assemble the program and go into the debugger to execute the program.

Type: ASM and press <RETURN>

Type: BUG and press <RETURN>

Type: <SHIFT><CLEAR>

Type: G600 and press <RETURN>

Did you see it? Probably not. This short assembly language program is executed so quickly, you can't even see the arrow displayed. There isn't even a noticable flicker on the screen.

Once the source code has been assembled to object code and the object code is stored in memory, how does the computer go about executing the program? You may remember from the Machine Architecture Module that the CPU can only execute one instruction at a time. To compensate for this the program is stored in memory and the CPU "fetches" one instruction at a time from memory. The CPU goes through a repeated cycle of fetching instructions one at a time and executing them until the entire program has been completed. The actual set of steps the microprocessor takes to execute a program is called the "fetch cycle."

Fetch Cycle

1. Fetch an instruction from memory. Get the opcode and an accompanying operand if there is one.

2. Advance the program counter to the address of the next instruction to be executed.

3. Execute the instruction.

4. Return to #1 and repeat the cycle.

First, the CPU fetches the instruction to be executed. Before executing the instruction, however, the CPU advances the program counter, a two byte register in the CPU, to the address of the next instruction to be executed. Then the CPU executes the instruction it had previously fetched. When the first instruction is completed, the CPU starts the cycle over again. The program counter holds the address of the next instruction to be executed. The next instruction is fetched and the program counter is advanced again. Read along as we execute the fetch cycle with the ARROW program.

1. Fetch the instruction. The CPU fetches the first instruction of the program from memory. It knows where the first instruction is, because you gave it the starting address of the program when you typed "G600". When the CPU fetches the instruction from memory, it gets both the opcode and the operand. In the ARROW program the CPU fetches both A9 and 7D. The opcode A9 is the signal to the CPU to also fetch the value (7D) in the next memory location. Opcodes not only instruct the CPU on what type of operation to perform, they also indicate to the CPU how many bytes in memory are associated with that instruction. This will become clearer as you proceed through the module. Look at Diagram 1 below. The CPU is holding the A97D (LDA #$7D) command.

2. Advance the program counter. Before the A97D (LDA #$7D) is executed, the program counter must be advanced to the address of the next instruction to be executed. The next instruction of the ARROW program is the 8D (STA), which is in memory location $602. Put the address of the 8D instruction in the program counter in Diagram 1.

3. Execute the instruction held in the CPU. Now execute the load command (A97D). Load the accumulator in Diagram 1 with $7D.

4. Return to #1 and repeat the cycle. Continue with the explanation of the fetch cycle below.

Diagram 1

```
Source Code      Object Code           6502 Processor

  *=$0600
  LDA #$7D       $600 | A9 |           COMMAND  [ A97D ]
                 $601 | 7D |
  STA $9C40      $602 | 8D |           PROGRAM COUNTER [      ]
                 $603 | 40 |
                 $604 | 9C |           ACCUMULATOR [    ]
  BRK            $605 | 00 |
```

1. Fetch the next instruction. The CPU fetches the next instruction based on the address in the program counter. The program counter has $602, so the CPU fetches the 8D (STA) instruction. This time the CPU fetches the two bytes in memory following the 8D in order to get the entire "store" command (STA $9C40). The 8D was a signal to the CPU that the instruction was a store instruction and that the operand was two bytes. The reason the operand is two bytes in this case is that the operand is the address of screen RAM ($9C40) and all addresses are two bytes. Thus, two more bytes are fetched from memory. You may have noticed that the two bytes of the address have been reversed, so that the low order byte, 40, is stored in memory before the high order byte, 9C. At this point it is not necessary for you to understand why the CPU does this. Just remember that whenever an address is stored in memory, the two bytes of the address are reversed. If you look at Diagram 2 below, you will see that the CPU holds the entire store command (8D409C).

2. Advance the program counter. The next instruction in the ARROW program is BRK (00). Place the address of the opcode 00 in the program counter in Diagram 2 before executing the previously fetched instruction.

3. Execute the instruction. Now the "store" command in the CPU is executed. In the Diagram below execute the instruction by storing the value in the accumulator in $9C40. When the arrow is stored in screen RAM, it appears on the screen.

4. Return to #1 and repeat the cycle. Continue with the last fetch cycle of executing the ARROW program below.

Diagram 2

```
Source Code      Object Code        6502 Processor

  *=$0600
  LDA #$7D       $600 | A9 |        COMMAND  [ 8D409C ]
                 $601 | 7D |
  STA $9C40      $602 | 8D |        PROGRAM COUNTER [        ]
                 $603 | 40 |
                 $604 | 9C |        ACCUMULATOR [ 7D ]
  BRK            $605 | 00 |
                       \    \
                       /    /
                       \    \
                $9C40 |    |
```

1. Fetch the next instruction. The address in the program counter is $605, so the opcode for BRK in $605 needs to be fetched. BRK is an instruction that does not require an operand. Consequently, the CPU only fetches one byte. The command the CPU fetches will always be one, two, or three bytes long. The CPU knows how many bytes to fetch from memory based on the opcode of the instruction. Place the opcode for the BRK instruction in the command box in the 6502 in Diagram 3 below.

2. Advance the program counter. The program counter is advanced to the address of the memory location following the BRK instruction where another instruction would be stored if there were more instructions in the program.

3. Execute the instruction. The BReaK instruction terminates the program. When a BRK instruction is executed, the address in the program counter is displayed, followed by the contents of the registers.

Diagram 3

Source Code	Object Code		6502 Processor	
*=$0600				
LDA #$7D	$600	A9	COMMAND	
	$601	7D		
STA $9C40	$602	8D	PROGRAM COUNTER	
	$603	40		
	$604	9C	ACCUMULATOR	
BRK	$605	00		

The computer is truly an amazing machine, but let's see if we can trick it by putting the value of an opcode into the position of an operand. Turn to Assembly Language Programming Worksheet #2.

Assembly Language Programming Worksheet #2

You will need an Assembler Editor Cartridge and an Advanced Topics Diskette to complete this worksheet and all the remaining worksheets in this module.

1. Boot up the system and ENTER the ARROW program.

   Type: ENTER #D:ARROW and press <RETURN>

2. LIST the program and then assemble it.

   Type: LIST and press <RETURN>

   Type: ASM and press <RETURN>

3. Note that the object code is listed by commands. So the two bytes for the LDA #$7D command are listed on one line (600 A97D). The next line contains the three bytes for the entire STA $9C40 command (602 8D409C). And the one byte for the BRK command appears on the last line of the object code (605 00). When the A9 is in the position of the opcode, which is the first byte of the command, the computer knows that the A9 represents a load the accumulator instruction. The computer also knows that the opcode is followed by a one byte operand. However, what would happen if you put an A9 in the position of an operand (eg. LDA #$A9)?

4. LIST the program again. Use the <CTRL> key in conjunction with the arrow keys to place the cursor over the 7 in the LDA #$7D command. Replace the 7D with A9.

   Type: A9 and press <RETURN>

   Press <BREAK> a few times to get below the listing of the program before assembling the program.

5. Assemble the program.

   Type: ASM and press <RETURN>

   The first line of the object code should read: 600 A9A9. Memory location $600 holds the first A9. The first A9 is the opcode for the LDA instruction. What will the computer do with the A9 in the operand? Follow the steps listed below to run the program.

Type: BUG and press <RETURN>

Type: <SHIFT><CLEAR>

Type: G600 and press <RETURN>

When you run the program, you should see an inverse "I". A9 is the internal character set code for that letter.

When a value is in the position of an instruction in the object code, the CPU treats the value as an instruction. Conversely, when the value is in the position of an operand in the object code the computer treats the value as an operand. In this program the operand is used as a letter to be printed on the screen. Thus, the opcode A9 tells the computer to load the accumulator with the value in the operand, which also happens to be an A9, and represents an inverse "I".

```
Opcode  Operand
    ↓  ↙
0600 A9A9    0110      LDA #$A9    ;LOAD ACCUMULATOR
0602 8D409C 0120      STA $9C40   ;STORE A9 ON SCREEN
```

Assembly Language Format

You have undoubtedly noticed that the source code of assembly language programs has a unique and structured format. The source code contains information in columns or "fields." There are three fields: the label field, the command field, and the comment field. Each field is separated from the next with a space. The label field and the comment field are optional.

Source Code Fields

Label	Command	Comment
BEGIN	LDA #$0D	;LOAD ACC. WITH A DASH

The Label Field

A label enables the programmer to assign a name to a command or to the beginning of a subroutine. A label must begin with a letter (A-Z), and it can only contain letters, numbers, and periods. It is good practice to make labels descriptive, but also try to limit them to no more than eight characters.

Suppose we put the label BEGIN in front of the first instruction in an assembly language routine which is similar to the ARROW program. And instead of having a BRK instruction at the end of the program, we replace it with a JMP instruction. A JMP instruction enables you to "jump" to a label. Look over the listing below. What do you think the program will do?

```
        *=$0600     ;ORIGIN AT $600
BEGIN   LDA #$0D    ;LOAD ACC. WITH A DASH
        STA $9C40   ;STORE IN SCREEN RAM
        STA $9C41   ;NEXT SCREEN LOCATION
        LDA #$7F    ;LOAD ACC. WITH >
        STA $9C42   ;STORE > ON SCREEN
        JMP BEGIN   ;REPEAT THE PROGRAM
```

The assembled version of the program is listed below.

```
                 10 ;                POINTER
                 20 ;
                 30 ;A PROGRAM TO DISPLAY TWO DASHES
                 40 ;AND A GREATER THAN SIGN IN THE
                 50 ;UPPER LEFT CORNER OF THE SCREEN
                 60 ;
                 70 ;
0000             0100        *=   $0600      ;ORIGIN AT $600
0600 A90D        0110 BEGIN  LDA  #$0D       ;LOAD ACC. WITH DASH
0602 8D409C      0120        STA  $9C40      ;STORE IN SCREEN RAM
0605 8D419C      0130        STA  $9C41      ;NEXT SCREEN LOCATION
0608 A91E        0140        LDA  #$1E       ;LOAD ACC. WITH >
060A 8D429C      0150        STA  $9C42      ;STORE > ON SCREEN
060D 4C0006      0160        JMP  BEGIN      ;DISCONTINUE PROGRAM
```

Note that the object code for the jump instruction holds the opcode for the jump (4C), and the address of the instruction which is accompanied by the label BEGIN. The assembler is responsible for assigning addresses to labels. The assembler goes through two steps to assemble your source program. When you type ASM, first the assembler reads through the source code and assigns memory addresses to each of the constants, variables, and labels. In this step a "symbol table" of the addresses and labels is compiled and stored in memory. The assembler allocates an area of memory just for this purpose. Then the assembler makes a second pass over your program and converts the source code to object code. Whenever the assembler encounters a label in the operand field, like JMP BEGIN, it inserts the label's address in the object code. Some assemblers provide a listing the symbol table after a program is assembled. The Atari assembler does not list the symbol table.

Look back at the listing above. Note that there are the two STA instructions in a row. When a STore the Accumulator instruction is executed, a copy of the accumulator is made in the specified location. The contents of the accumulator is not affected by the execution of a STA instruction. The accumulator remains unaltered. Thus we can use a second STA instruction to store the same character in another location on the screen. Turn to Assembly Language Programming Worksheet #3 to see how to insert a label into a program and observe what this new program does.

Assembly Language Programming Worksheet #3

1. ENTER the POINTER program on the Advanced Topics Diskette.

   Type: ENTER #D:POINTER   and press <RETURN>

2. LIST the program. The listing displays a series of load and store instructions, terminated by a BRK instruction. First you will insert a label on the first line of the program. Use the <CTRL> and arrow editing keys to place the cursor directly over the space before the LDA instruction.

3. While holding down the <CTRL> key, press the <INSERT> key (in the upper right hand corner of the keyboard) five times - once for each letter in the word BEGIN.

   Type: BEGIN   and press <RETURN>

   Be sure there is a space between the label and the command. If not, repeat steps one through three.

4. Using the <CTRL> and arrow editing keys again, move the cursor down over the "B" in the BRK instruction.

   Type: JMP BEGIN   and press <RETURN>

Your listing should look like this. Try editing the comment on line 130.

```
0100  *=$0600            ;ORIGIN AT $600
0110 BEGIN LDA #$0D      ;LOAD ACC. WITH DASH
0120  STA $9C40   ;STORE ON THE SCREEN
0130  STA $9C41   ;NEXT SCREEN LOCATION
0140  LDA #$7F    ;LOAD ACC. WITH >
0150  STA $9C42   ;STORE ON THE SCREEN
0130  JMP BEGIN   ;DISCONTINUE PROGRAM
```

The numbers on the left are the decimal line numbers. They are there strictly for editing purposes. When you are in the editor you can delete or insert lines using the specific line numbers.

5. Assemble and run the program.

   Type: ASM   and press <RETURN>

Type: BUG and press <RETURN>

Type: G600 and press <RETURN>

6. You have created an infinite loop. The program is continually repeating itself. This time you didn't have to type <SHIFT><CLEAR> before running the program because the infinite loop prevents the screen from scrolling. To stop the program you must press the <BREAK> key.

The label field is always separated from the command field with a space. If no label is being used, you must leave a space between the line number and the command field. The space indicates to the assembler that no label is used on that line.

The Command Field

The "command" field follows the label field. The command field includes the instruction and the operand. The three-letter instructions are also referred to as "mnemonics." Mnemonic means memory device or aid for remembering. Assembly language instructions are three-letter abbreviations for the operation that will be performed, thus they help us remember what the instruction does.

$$\overbrace{\text{mnemonic operand}}^{\text{Command Field}}$$

LDA #$7D

There is always one space between the mnemonic and the operand in the command field.

The Comment Field

The third field is the "comment" field. Comments are optional but highly recommended. You will find in assembly language programming that even though you may know a program inside and out when you write it, when you go back to it a few days later, you will struggle to remember exactly how the program works if the code is not well documented.

Comments are separated from the other fields with a ";". Comments can follow the command field or you can start a line with a ";" and devote the entire line to a comment.

```
80    ;          Character Display
90    ;
0100 ;THIS PROGRAM PRINTS WHATEVER CHARACTER
0110 ;IS STORED IN THE ACCUMULATOR ONTO THE
0120 ;GRAPHICS 0 SCREEN.  THE VALUES FOR
0130 ;THE INTERNAL CHARACTER SET ARE USED
0140 ;TO STORE A CHARACTER IN SCREEN RAM.
0150 ;
0160  *=$0600           ;ORIGIN AT $600
0170 BEGIN LDA #$1E     ;LOAD ACC. WITH A >
0180  STA $9C40         ;SCREEN RAM
0190  JMP BEGIN         ;REPEAT PROGRAM
```

As long as comments are preceeded with a ";", a comment can contain anything, (letters, numbers, symbols,etc.) just like comments following a REM statement in BASIC. When the assembler converts the source code to object code, the comments are ignored.

Psuedo Opcodes

You have probably also noticed that the first line of every assembly language program you have seen thus far contains an "*" followed by an "=" and an address (usually $0600). In assembly language you must tell the assembler where in memory to store the object code of your program. The Atari uses an asterisk to set the starting address of the program's object code in memory, which is referred to as the "origin" of the program. The equals sign is a "psuedo opcode." A psuedo opcode is an instruction to the assembler. For example, "*=$0600" instructs the assembler to set the origin of the program equal to $600. Psuedo opcodes are not translated into 6502 object code. They are instructions to the assembler. Turn to Assembly Language Programming Worksheet #4 to change the origin of the POINTER program.

Assembly Language Programming Worksheet #4

1. ENTER the POINTER program on your Advanced Topics Diskette into memory.

   Type: ENTER #D:POINTER and press <RETURN>

2. LIST the program. Use the <CTRL> and the arrow keys to move the cursor up over the first "0" in the address "$0600" on line 0100.

3. While holding down the <CTRL> key, press the <DELETE> key once. The DELETE key is in the upper right hand corner of the keyboard. The cursor should now be sitting over the "6" in "$600". You have deleted the first 0.

4. Now use the <CTRL> and the arrow keys to move the cursor to the space just past the last "0" in "$600". You are going to change the starting address of the object code from $600 to $6000.

   Type: 0 and press <RETURN>

   Edit the comment on line 0100 too. The first line of your program should look like the following.

```
0100  *=$6000     ;ORIGIN AT $6000
```

5. Press the <BREAK> key a few times to move the cursor down below the program. Now assemble the program.

   Type: ASM and press <RETURN>

6. Look closely at the addresses of the object code. They no longer start with 600. The object code is stored in memory starting at $6000 instead. Also, even though the first line of your program was "*=$6000", the first byte of the object code is A9, for the LDA instruction. The "*=" psuedo opcode is an instruction to the assembler. The 6502 never translates psuedo opcode instructions to machine language as part of the object code.

7. Go into the degugger by typing BUG to run the program. What instruction will you use to execute this program? (Hint: "G" stands for go, and the number which follows is the origin or starting address of the program in memory.) Run the program.

Up to this point we have been storing the object code of the assembly language programs on page six of memory ($600-$6FF). Page six is a free area of RAM and a good place for short assembly language programs. As your programs get longer you can set the origin of your program to any address in the free RAM area between $2000-$A000. However, if you are using $9C40 - $A000 for screen RAM, as we are throughout this module, you should probably originate your program between $2000-$9000. The area starting at $6000 is good for programs which are too long for page 6 storage.

If you plan you integrate your assembly language program with a BASIC program or a commercial utility program, bare are in mind that page six of memory is used quite frequently by commercial software. Also, if you use the USR function to run an assembly language program from BASIC, you need to avoid having one program write over another program in memory. The area of memory starting at $6000 tends to be a safe and spacious area for your routines.

In assembly language it is possible to give a name to an address that you use in your program. For example, instead of using the address $9C40, we could assign the name SCREEN to the address. Then any time we wanted to store a value at that address, we could just use the name SCREEN. Using labels rather than listing a hexadecimal addresses in the operand makes assembly language programs much easier to read and understand. To assign a name to a variable or an address, we must use the "=" psuedo opcode.

Constant and variable declarations are grouped together in assembly language programs and commonly follow the origin statement at the beginning of the program. Take a look at the example below.

```
 *=$0600             ;ORIGIN AT $600
SCREEN = $9C40       ;GR.0 SCREEN START LOCATION
 LDA #$7D            ;LOAD ACC. WITH AN ARROW
 STA SCREEN          ;STORE IN SCREEN RAM
 BRK                 ;DISCONTINUE PROGRAM
```

Note that the "S" in SCREEN is in the label field. All variable and constant declarations begin in the label field, one space before the command field.

As this program is expanded, any time you want to refer to the address, $9C40, you just use the label SCREEN. Using constant and variable names makes a program much easier to read and understand. Also, whenever you go to change the address you are using, all you need to do is change the constant declaration at the beginning of the program. From then on the assembler treats the word SCREEN as the new address. Otherwise, you need to search through your program to find every instance in which you used the address $9C40. As your assembly language programs get longer, locating all the instances of $9C40 becomes an extremely arduous task. To experiment with assigning a label to an address and then changing that address, turn to Assembly Language Programming Worksheet #5.

Assembly Language Programming Worksheet #5

1. ENTER the POINTER program on your Advanced Topics Diskette.

   Type: ENTER #D:POINTER and press <RETURN>

2. LIST the program. To insert a statement that assigns the label SCREEN to $9C40, you must add another line to the program.

```
Type: 0105 SCREEN = $9C40     and press <RETURN>
            \     / /
             Space
```

   LIST the program to see that the line has been added.

3. Now replace the screen address in the STA instruction with the word SCREEN. Using the <CTRL> and the arrow keys, move the cursor up and place it over the "$" in the STA $9C40 instruction on line 120.

   Type: SCREEN and press <RETURN>

4. Assemble the program, go into the debugger, and execute the program. Assigning a name to the screen address should not have affected the operation of your program in any way.

The addresses for the second dash and the greater than symbol are still listed in hexadecimal on lines 130-150. Suppose we used the label SCREEN to make the program more readable. However, each of the addresses for the screen is one greater than the previous screen address, in order to display the dash and the arrow in subsequently screen locations. There is one option we could use in this case, which would enable us to use the label SCREEN while also incrementing the screen locations. Add one to the label SCREEN in the operand. Look over the example below.

```
 *=$0600              ;ORIGIN AT $600
SCREEN = $9C40        ;START GR.0 SCREEN
 LDA #$0D             ;LOAD ACC. WITH A DASH
 STA SCREEN           ;DISPLAY ON THE SCREEN
 STA SCREEN+1         ;NEXT SCREEN LOCATION
 BRK                  ;DISCONTINUE PROGRAM
```

In the example above, the label SCREEN is used again, and a one is added to it in the operand. When the STA instruction is executed, the processor will add the one to the address of SCREEN ($9C40) and store the contents of the accumulator at the new address ($9C41).

5. Use the <CTRL> and the arrow keys to move the cursor to the "$" preceeding the screen address in line 130. Replace the hexadecimal address with SCREEN+1.

6. Now add two to the SCREEN address in line 150. Use the <CTRL> and the arrow keys to move the cursor to the "$" preceeding the address $9C42. Type in SCREEN+2 and press <RETURN>.

7. Assemble the program. If you get an error message, LIST the program and check to see that all of your fields line up with one another.

8. Go in to the debugger by typing BUG and pressing <RETURN>. Run the program from $600 by typing G600. The performance of the program should be the same. The progam is much easier to read now though.

9. Experiment with changing the addresses of screen RAM you are using. The addresses for the screen range from $9C40 to $9FFF. Use the <CTRL> and the arrow keys to put the cursor over the addresss in the SCREEN = $9C40 assignment. Change the address. Be sure to press <RETURN> after typing in a new address and move the cursor down below the program before trying to assemble it. Can you put the arrow in the middle of the screen?

For purposes of explanation, the address of screen RAM will be used instead of the name SCREEN in the next couple of programs. In your own programs you should avoid using hexadecimal values in the operand. Use labels wherever possible.

6502 Assembly Lanuguage Instruction Set

The most commonly used assembly language instructions are explained and demonstrated in this section. Five of the addressing modes used in assembly language also are introduced.

There are 56 instructions in the Atari 6502 instruction set. Each instruction consists of a three-letter mnemonic, which is an abbreviation for the operation the instruction performs.

The most common instructions are those that transfer data between the microprocessor and memory. All the data transfers that go on between the CPU and memory involve one of the internal registers. "Load" instructions transfer memory data into the accumulator, the X register, or the Y register. There are three load instructions - one for each register.

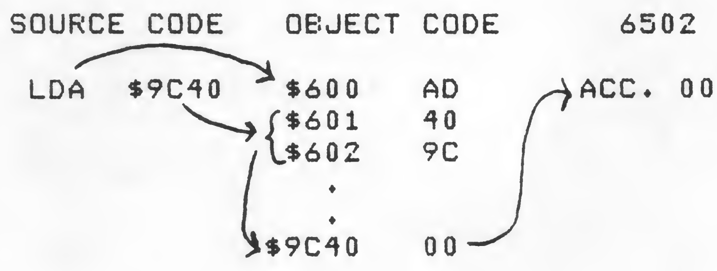
LDA:	<u>L</u>oa<u>D</u> the <u>A</u>ccumulator
LDX:	<u>L</u>oa<u>D</u> the <u>X</u> Register
LDY:	<u>L</u>oa<u>D</u> the <u>Y</u> Register

You are familiar with the LDA instruction.

$$\begin{tikzcd}
\underline{\text{SOURCE CODE}} & \underline{\text{OBJECT CODE}} & & \underline{\text{6502}} & \\
\text{LDA \#\$0D} \arrow[r, bend left] \arrow[dr, bend right] & \text{\$600} & \text{A9} & \text{ACC.} & \text{0D} \\
& \text{\$601} & \text{0D} \arrow[ur] & &
\end{tikzcd}$$

The value in memory immediately following the opcode for the LDA instruction is stored in the accumulator. The "#" is referred to as an "immediate" symbol. The LDA #$0D command is read, "load the accumulator with an immediate hexadecimal $0D." Whenever you use a hexadecimal number, you must precede the value with a "$". To use decimal numbers in a program, simply list the decimal amount and forgo with the dollar sign. LDA #13 is the same as LDA #$0D, since decimal 13 equals hexadecimal $0D. The "#" remains because we are still loading the accumulator with the value immediately following the instruction. The load instructions for the X and Y registers function exactly the same way. LDX #$0D places hexadecimal $0D in the X register. LDY #$0D places a hexadecimal $0D in the Y register. Loading a register with a specific value is called "immediate addressing." Immediate addressing is easily recognized by the "#" preceding the value to be loaded into the register.

It is also possible to load a register with the contents of a memory location. Suppose you have a program that computes a math problem and stores the answer in memory. When the program is done, you don't know what the answer is, but you do know where the answer has been stored. You need to be able to load a register with the contents of the address of the answer, so you can find out what the answer is. Loading a register with the contents of a memory location is called "absolute addressing." In absolute addressing, the operand to the instruction is the address of the memory location you wish to see. Study the diagram below to see how absolute addressing works.



The zero stored in $9C40 is loaded into the accumulator. Since this is absolute addressing, the "#" is no longer used. Note that the opcode for the LDA instruction stored in $600 is "AD". Up until now the opcode for LDA has been A9. The opcode changed because the operation performed by the CPU is different. AD instructs the CPU to get the value stored in the specified memory location and load it into the register. The AD also instructs the CPU to fetch two additional bytes, for the address in the operand. You needn't worry about what the specific values are of the various opcodes, or which opcodes represent which addressing modes. The assembler and the processor handle that for you. Our goal here, is to point out that the opcode indicates to the CPU the type of addressing being used and thus, what operation the CPU is to perform. When the processor encounters the AD, it knows that it must fetch an address from memory, and load the accumulator with the contents of that address.

Assembly Language Programming Worksheet #6

Turn off the computer and reboot your system to begin this worksheet. It is necessary for you to start the exercise with memory and the registers cleared.

1. ENTER and LIST the ATSIGN program.

2. Note the LDA instruction on line 120. The instruction reads, "load the accumulator with an immediate decimal 32." What number will be stored in the accumulator? ________

Assemble the program. Then go into the debugger and press <SHIFT><CLEAR> to clear the screen. Run the program (G600). When the program stops, the registers will be listed. Were you right? When the assembler translates the source code to object code the decimal values are listed in hexadecimal.

3. Type X to go back to the editor and LIST the program. Now change the LDA #32 instruction to LDA #298. What will be loaded into the accumulator? ________ Assemble the program.

That was a trick question. You should have gotten Error 10. Page 43 of the Assembler Editor Manual lists the error messages. Error 10 states, "the expression is greater than 255 where only one byte is required." Remember that one memory location holds a maximum of 255. If you try to load a number greater than 255 into the accumulator, the program will not assemble.

4. Now try absolute addressing. LIST the program. On line 110, replace LDA #298 with LDA $600. What value will be loaded into the accumulator? ________ If you are unsure, assemble the program and then try to answer the question. The object code for the LDA instruction should appear as follows.

```
0600  AD0006   0110    LDA $600
```

LDA $600 loads the accumulator with the contents of memory location 600. What is the value in $600 which will be loaded into the accumulator? ______

5. Run the program from the debugger (G600). The contents of the registers will be displayed after the program is executed. Check the contents of the acuumulator against your answer.

6. Define the addressing modes used below and explain what the instruction will do.

LDA #$7D ____

____

____

LDA #64 ____

____

____

LDA $9C40 ____

____

____

LDA SCREEN ____

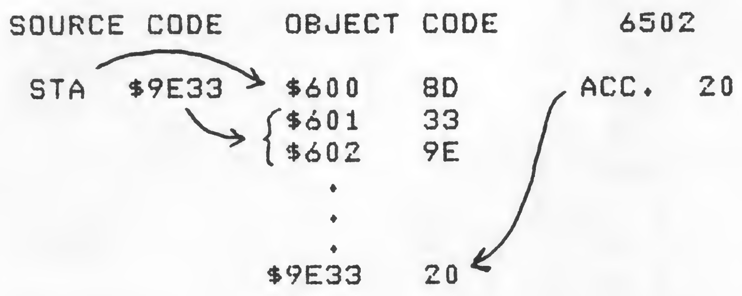
____

____

Whenever you want to put a value in memory, you use a "store" command. There are three store instructions - one for each register.

STA:	STore the value in the Accumulator in memory.
STX:	STore the value in the X register in memory.
STY:	STore the value in the Y register in memory.

In the ATSIGN program the STA instruction was used to put the value for an at sign into memory location $9E33 which had been assigned the label SCREEN. This is another example of absolute addressing.



The $20 in the accumulator is stored in memory location $9E33. Actually, a copy of the $20 is made and stored in $9E33. The $20 in the accumulator remains unaffected by the STA command. Turn to Assembly Language Programming Worksheet #7 to try the different load and store instructions.

<u>Assembly Language Programming Worksheet #7</u>

You will need to turn off your machine and reboot the system with an Assembler Editor Cartridge and the Advanced Topics Diskette in order to clear the registers.

1. ENTER and LIST the ATSIGN program.

2. Use the editing keys to place the cursor over the A in the LD<u>A</u> instruction. Instead of loading the accumulator with #32, load the X register with #32. Type an X to replace the A.

3. If the value for the at sign is being loaded into the X register, then to print the at sign on the screen, you must store the contents of the X register in screen RAM ($9E33). Change the STA command to a ST<u>X</u> command.

4. Assemble the program. Type BUG to get into the debugger. Type <SHIFT><CLEAR>, to clear the screen, and run the program from $600 by typing G600.

5. The contents of the internal registers will be listed on the screen once the program is completed. List the contents of the different registers below.

A=   X=   Y=

As you can see, the program's performance does not change by using the load and store instructions for the X register. However, now the value for the at sign is stored in the X register instead of in the accumulator. Now let's see where the #20 ends up when the program is executed.

Type: D9E33   and press RETURN

The "D" stands for display. You are displaying the contents of memory location $9E33.

You should see a 20. A copy of the 20 in the X register has been stored in $9E33.

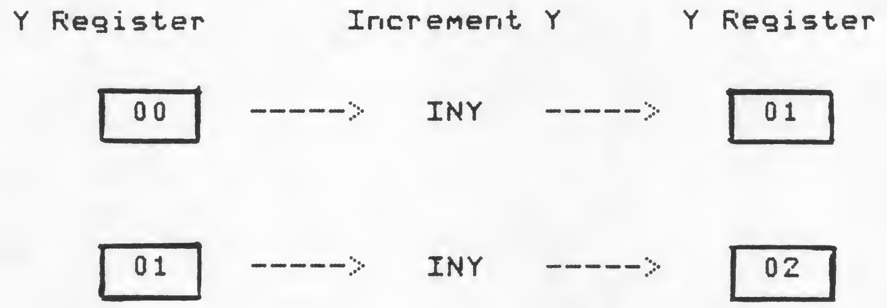
You have seen how the assembler translates the source code to object code, and you are familiar with the format of assembly language programs and how they are executed. Now let's get on with some assembly language programming. In one example that you saw, a short assembly language program which placed an arrow on the screen, was executed so quickly that you couldn't even see the arrow displayed. The alternative program that we used leaves the character on the screen. What good is assembly language if we can't control how long something will be displayed on the screen? What we need is a "delay loop," which acts as a timer. Suppose we put a character on the screen and then set a timer to count to 255. While the character is being displayed on the screen, the timer ticks away. When the timer gets to 255, the program will continue with the next instruction.

To simulate a timer with a delay loop, we need to use an "increment" instruction. Increment instructions add one to a counter. There are three increment instructions.

INC: Add one to the contents of a memory location.
INX: Add one to the contents of the X register.
INY: Add one to the contents of the Y register.

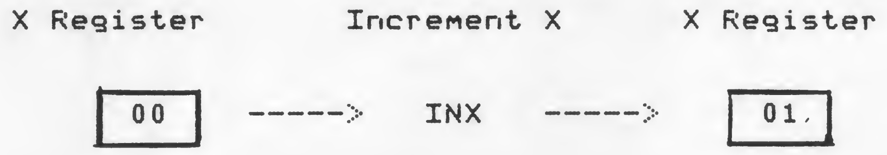
Note that there is no increment instruction for the accumulator. The INC instruction will be explained later.

The diagram below illustrates how the INY, INcrement the Y register, instruction works.



The 6502 handles the addition for you and stores the new value in the Y register.

The X register can be incremented in the same way with the INX instruction.



The INX and INY instructions are self-sufficient commands. There is no operand necessary for INY or INX. When an instruction contains all of the information the CPU needs, it is called "implied addressing." The operation to be performed is implied by the INY and INX instructions.

```
*=$0600         ;ORG program at $600
LDY #00         ;LOAD Y WITH 0
INY             ;ADD ONE TO THE VALUE IN Y
BRK             ;BREAK
```

BRK is another example of an instruction that uses implied addressing. It does not require an operand. The CPU understands from the BRK instruction alone that it is to discontinue execution of the program.

It is not possible to increment the accumulator. Instead, the third increment instruction enables you to add one to the contents of a memory location. For example, suppose you have a variable called COUNTER in your program and it is stored in memory location $CD. $CD is a free memory location on the zero page of memory. Look over the program below to see how to use the INC instruction to add one to COUNTER.

```
 *=$0600             ;ORIGIN at $600
COUNTER = $CD        ;ASSIGN COUNTER TO LOCATION $CD
 LDA #00             ;LOAD ACC. WITH 0
 STA COUNTER         ;INITIALIZE COUNTER
 INC COUNTER         ;ADD ONE TO THE VALUE IN COUNTER
 BRK                 ;BRK
```

COUNTER is initially set to 0. When the INC COUNTER instruction is executed, one is added to the value stored in COUNTER. It is also possible to place an address in the operand of an INC instruction. For example, in the program above, INC $CD would have served the same function as INC COUNTER. However, using variable names is preferred. Variable names make programs more understandable both to the programmer and anyone else reading the program. Variable names also enable you to easily alter or update a program. To experiment with the increment commands turn to Assembly Language Programming Worksheet #8.

Assembly Language Programming Worksheet #8

To begin this worksheet, you will need to turn off your computer and reboot the system with an Assembler Editor Cartridge and the Advanced Topics Diskette. This will clear all the registers.

1. You should have the EDIT prompt on the screen. Type in the following program. Be sure to leave two spaces between the line number and the instruction. Press <RETURN> after entering each line.

```
100  *=$600
110  LDY #$A0
120  INY
130  BRK
```

2. After running this program, what number would you expect to find in the Y register?_______ Execute the program from the debugger and see.

3. To get back to the editor,

   Type: X   and press <RETURN>

4. LIST the program. Now try experimenting with incrementing different values in the Y register. Using the editing keys, place the cursor over A in the value to be loaded into the Y register ($A0). Replace the number with the values listed below. Fill in the boxes with the new values held in the Y register after executing the program.

Y Register				Y Register
09	----->	INY	----->	
FE	----->	INY	----->	
FF	----->	INY	----->	

When you incremented $FF, you should have gotten 00 in the Y register. $FF is the largest byte or two digit hexadecimal number. When one is added to $FF, the sum is $100.

```
$ FF
 +01
$100
```

Similarly, in base 10, 99 is the largest two digit number that can be represented. Adding one to 99 resets the two digits to 0 and carries a one over into the next place value. Since the registers and memory locations in the Atari only hold one byte, when one is added to $FF, the Y register is reset to zero.

5. Step through the last program which increments $FF. Type BUG to get into the debugger. From the debugger,

Type: S600 and press <RETURN>

First, the LDY #$FF instruction is executed and the Y register is set to $FF.

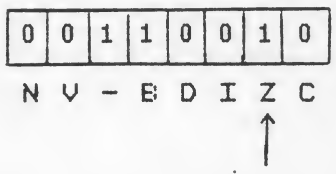
Type: S and press <RETURN>

This time the INY instruction is executed. At the bottom of the screen you should see the following. (Don't worry if the S, stack pointer, on your display does not equal 08.)

```
     0602       C8          INY
          A=00   X=00   Y=00   P=32   S=08
```

The "P=" stands for the processor status register. The status register is one of the internal registers in the 6502. The status register holds one byte, however, each bit holds significant information concerning the results of the CPU's most recently executed instruction. For example, if the last instruction left a negative number in one of the registers, the negative bit of the status register would be set. (The status register was first introduced in the Machine Architecture Module. See the Central Processing Unit section if you need to review.) Each bit of the status register is called a flag. The flags indicate if a certain condition exists in the processor. Currently, the status register on your screen should hold 32 (P=32). The binary representation of the status register below shows the bit pattern for the hexadecimal number $32. The ones indicate which bits of the status register are set or turned on.

Status Register



The "Z" bit, or zero flag, is set. The result of the last instruction (INY) left a zero in the Y register, and consequently the zero flag of the status register was set. The "-" or unused bit and the "B" or the break bit were also set. The unused bit is set as a program is executed and the break bit was set by the BRK instruction at the end of the program. The importance of the status flags will become clearer in the next section. Don't worry if you find them a bit confusing. The status register is typically difficult for beginners to understand.

Assembly Language Programming Worksheet #9

There is also a set of "decrement" instructions. Decrement instructions are the opposite of increment instructions.

DEX subtracts one from the value in the X register.

DEY ______________ one from the value in the ___ register.

DEC subtracts one from the contents of a memory location. DEC COUNTER subtracts one from the value stored in COUNTER.

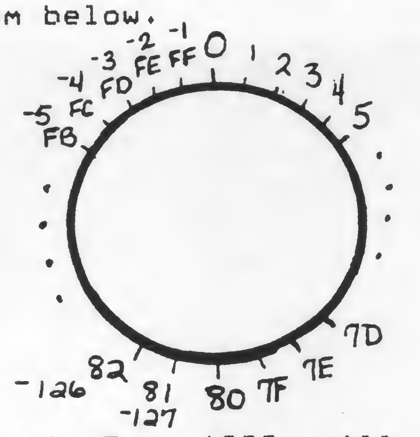
1. Use the editing keys to change the increment command in the increment routine, which you used in worksheet #8, to a decrement instruction as listed below. If you no longer have the increment program in memory, type in this new program.

```
100   *=$600
110   LDY #$FF
120   DEY
130   RTS
```

2. Assemble the program and run it from the debugger. Try the different values for the Y register listed below. Fill in the boxes on the right with the results of the DEY instructions.

Y Register				Y Register
FF	---->	DEY	---->	
F0	---->	DEY	---->	
00	---->	DEY	---->	

Decrementing 00 should have given you $FF in the Y register. In assembly language $FF stands for a minus one as well as 255. The CPU uses a circular number line. Take a look at the diagram below.



If you add one to $FE you get $FF. If you subtract one from zero, you also get $FF. $FF represents a minus one and 255 in the computer. You can tell if the $FF represents a minus or 255 by looking at the status register flags. When one is subtracted from 00, the result is $FF and the negative bit of the status register is set. When one is added to 255, the carry flag is set, indicating that the number has exceeded the amount which can be held in one byte.

2. Step through the last decrement program which subtracts one from zero.

Type: S600 and press <RETURN>

The contents of the registers will be listed as each instruction is executed. The Y register should hold 00, from the LDY #$00 instruction.

3. Type S and press <RETURN>, to execute the DEY instruction. The current contents of the registers will be listed. Fill in the registers below with what appears on your screen.

```
          0602      88      DEY
               A=       X=       Y=       P=       S=
```

4. The status register (P) should have "B0" in it after executing the DEY instruction. Remember that the status register holds the status flags. Each bit of the status register holds significant information. The binary bit pattern for B0 and the status flags associated with each bit are shown below.

Status Register

1	0	1	1	0	0	0	0
N	V	-	B	D	I	Z	C
↑							

The "N" or negative flag has been set to indicate that decrementing 00 resulted in a negative number (-1 or FF).

Don't worry if you don't understand the peculiar numbering system or the status register of the CPU just yet.

The increment and decrement instructions will prove to be very useful in setting up a delay loop. Now we need some way to repeat or "loop" through a set of instructions to create a delay. To write a loop that is repeated a specified number of times, we will use a "branch" instruction. For example, the "BNE" instruction stands for Branch Not Equal to zero. BNE can be used to repeat a decrement instruction, until the register reaches zero. Take a look at the short program below which uses a BNE instruction for a timing loop.

```
 *=$600
SCREEN = $9E33
 LDY #$FF      ;SET COUNTER
 LDA #$20      ;CODE FOR AN AT SIGN
 STA SCREEN    ;DISPLAY ON THE SCREEN
DELAY DEY      ;SUBTRACT 1 FROM Y
 BNE DELAY     ;IF Y IS NOT 0, DEY AGAIN
 BRK           ;TERMINATE PROGRAM
```

In the example above, as long as the Y register is not zero, the CPU will branch back to the label DELAY and decrement the Y register again.

To determine if the Y register has reached zero, the BNE instruction checks the zero flag of the status register. When the register is decremented to zero, the zero flag of the status register is set. When the BNE instruction finds that the zero flag of the status register is set, the condition for branching when the Y register is not equal to zero is no longer exists. The register is zero and so the branch is not taken. Instead, the next instruction in the program is executed.

The 6502 instruction set has a series of branch instructions, each of which checks the current condition of one of the status flags. You can branch on a negative number, a positive number, a carry, etc. Below are the eight branch instructions available with the Atari Assembler Editor.

BCC:	Branch on Carry Clear
BCS:	Branch on Carry Set
BEQ:	Branch on EQual to zero
BMI:	Branch on result MInus
BNE:	Branch Not Equal to zero
BPL:	Branch on result PLus
BVC:	Branch on oVerflow Clear
BVS:	Branch on oVerflow Set

Branch instructions are very useful for short distance branches, as is the case with timing loops. However, it is not possible to branch long distances in a program. In a large program where a long branch is needed, the alternative to a branch instruction is a "JSR", Jump to a SubRoutine. JSR will be explained in the next section.

Turn to Assembly Language Programming Worksheet #10 to see how increment, decrement, and branch instructions can be used in a delay loop to have more control over how long something is diplayed on the screen.

Assembly Language Programming Worksheet #10

1. ENTER the DELAY1 program on your Advanced Topics Diskette.

```
Type:  ENTER #D:DELAY1  and press  <RETURN>
```

```
             25 ;              DELAY
             50 ;
0000         0100         *=     $0600
9E33         0110 SCREEN  =      $9E33 ;SCREEN RAM
0600 A000    0120         LDY #$00     ;SET COUNTER
0602 A920    0130         LDA #$20     ;CODE FOR @
0604 8D339E  0140         STA SCREEN   ;DISPLAY @
0607 C8      0150 DELAY   INY          ;ADD 1 TO COUNTER
0608 D0FD    0160         BNE DELAY    ;IF NOT 0, REPEAT DELAY
060A A900    0170         LDA #00      ;LOAD ACC. WITH 0
060C 8D339E  0180         STA SCREEN   ;ERASE @
060F 00      0190         BRK          ;BREAK
```

2. LIST the program. It should look like the listing above. The Y register serves as a timer which counts to 255 while the at sign is being displayed on the screen. A blank space is displayed over the at sign as soon as the delay is completed. Consequently, we can see the results of the delay or the computer counting to 255.

3. Complete the following steps to execute the program.

```
Type:  ASM  and press <RETURN>

Type:  BUG  and press <RETURN>

Type:  <SHIFT><CLEAR>

Type:  G600  and press <RETURN>
```

You would think that because the computer has to count to 255, the at sign would stay on the screen longer, before it was erased. It doesn't look much different than the ARROW program did without a delay, does it? It is longer, though. Step through the program to see that the Y register is really being incremented 255 times while the at sign is on the screen. Do the following.

```
Type:  S600 and press <RETURN>
```

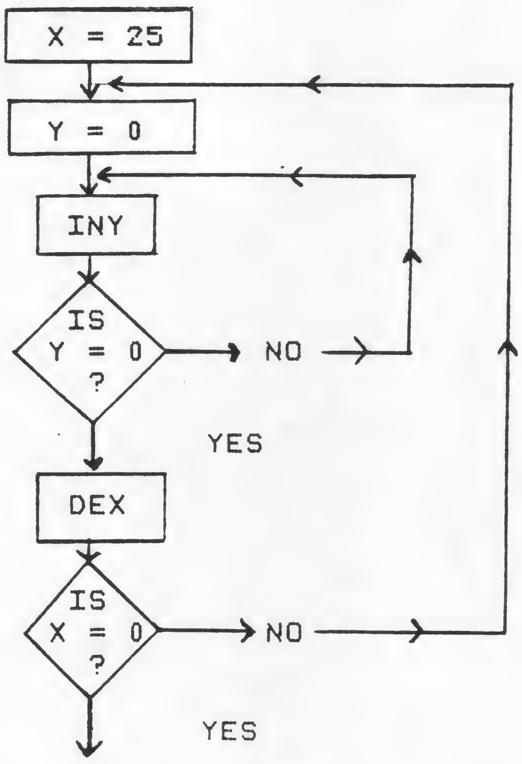
Continue to type S and <RETURN> a few times to see the Y register being incremented.

Branch instructions are always followed by a label. The label indicates where to branch to. Branch instructions use "relative addressing." The object code for a branch command is two bytes, one byte for the instruction, and one byte for the "offset," or the distance from the branch to the label. The offset is the number of bytes in memory between the branch instruction and the instruction accompanying the label you are branching to. Look at the object code for the branch command in the DELAY1 program below.

```
0607 C8      0150 DELAY  INY             ;ADD 1 TO COUNTER
0608 D0FD    0160        BNE DELAY       ;IF NOT 0, REPEAT DELAY
```

Memory location $608 holds, D0, the opcode for the BNE instruction. The FD in $609 is the offset to the label DELAY. FD, in this case, represents a decimal -3. The CPU must look back three bytes in memory to find the instruction associated with the label DELAY. Since the offset is one byte in the object code, the distance that is branched must be held in one byte. Consequently, you can branch up to 128 bytes forward ($00-$80), and 127 bytes back ($81-$FF) in a program and no further. Branch instructions are the only assembly language instructions that use relative addressing. The offsets in the object code are handled by the CPU. All you need to worry about is branching too far in your programs.

A longer delay is needed in order to leave the character on the screen for a longer period of time. To create a longer delay we will need to use another register. This second register will count the number of times the first register counts from 0 to 255. What we will do is "nest" the 0-255 timing loop inside another loop. Suppose we load the X register with 25 and each time the Y register counts from 0 - 255 the X register is decremented. This cycle is continued until the X register is zero.



Here is the assembly language version of the nested delay loops illustrated in the flowchart.

```
DELAY LDX #25     ;COUNT 25 Y LOOPS
AGAIN LDY #00     ;START WITH 0
 WAIT INY         ;ADD 1 TO Y
      BNE WAIT    ;IF NOT 0, REPEAT WAIT
      DEX         ;SUBTRACT 1 FROM X
      BNE AGAIN   ;IF NOT 0, REPEAT AGAIN
      BRK         ;BREAK
```

The delay loop is now a separate subroutine, which the ATSIGN routine will "call." The advantage of making the delay loop a separate subroutine is that it can be used from anywhere in an assembly language program. As you have seen, assembly language is processed so rapidly that delay loops are commonly needed. If the nested delay loop had been incorporated into the middle of the ATSIGN program, it could only be used when an at sign was being printed in the middle of the screen. The secret to good assembly language programming is to write versatile subroutines that can be reused within the program.

Turn to Assembly Language Programming Worksheet #11 to experiment with changing the length of the delay.

Assembly Language Programming Worksheet #11

1. ENTER the DELAY2 program on the Advanced Topics Diskette.

   Type: ENTER #D: DELAY2 and press <RETURN>

The listing of the program should look like this:

```
       *=$600
      SCREEN = $9E33
       LDA #$20          ;CODE FOR @
       STA SCREEN        ;DISPLAY ON SCREEN
       JSR DELAY         ;WAIT ROUTINE
       BRK               ;BREAK
;
;
;
      DELAY LDX #$A0      ;COUNTER FOR Y LOOPS
      AGAIN LDY #00      ;0-255 COUNT
      WAIT INY           ;ADD 1 to Y
       BNE WAIT          ;IF NOT 0,REPEAT WAIT
       DEX               ;SUBTRACT 1 FROM X
       BNE AGAIN         ;IF NOT 0, REPEAT AGAIN
       RTS               ;RETURN
```

The "JSR" instruction, which stands for Jump to the SubRoutine, is used to call the delay routine. The RTS instruction at the end of the delay routine tells the CPU to ReTurn from the Subroutine and go back to executing the instructions in the DELAY2 routine.

2. The value stored in the X register controls the length of the delay. Assemble the program and execute it from the debugger to see how long the delay lasts.

3. To return to the editor,

   Type: X and press <RETURN>

4. Use the <CTRL> and the arrow keys to replace the #$A0 in the LDX #$A0 command with #$F0. Be sure to press <RETURN> after completing your edit. Assemble and run the program from the debugger. What effect did changing the value in the X register have on the delay?

----

5. What would happen it you changed the value loaded into the X register to #$5?

Try it and see.

Summary

For a summary of the 6502 instructions explained thus far, see the table at the back of this module.

The 6502 offers eight different addressing modes. The addresssing modes that have been covered thus far are listed below.

Immediate   LDA #$7D

Perform the operation on the specified 8 bit value. The immediate symbol (#) must precede the value in the operand.

Absolute   STA $9C40

Perform the operation on the value stored in the specified memory location. The operand must contain an address or a label which represents an address.

Implied   INX, RTS

The operation to be performed is implied by the instruction itself.

Relative   BNE AGAIN

Relative addressing is used exclusively with branch instructions. The object code holds the offset which indicates the number of bytes in memory between the branch instruction and the destination of the branch.

Zero Page   LDA $CD

Perform the operation on the contents of the specified zero page address.

Zero page addressing is the same as absolute addressing, except that the address being accessed is on the zero page. Addresses on the zero page are listed as one byte because the high order byte of the address is "00". The complete address of $CD is $00CD. When zero page addressing is used, the object code for the command is only two bytes, one byte for the instruction, and one byte for the address. The CPU assumes that the high order byte of the zero page address is $00. Variables that are used frequently in a program are commonly stored on the zero page for quick and easy access.

Indexed Addressing Modes

The three indexed addressing modes used in 6502 assembly language are explained in this section. Two of the three indexed addressing modes will be used in the final animation program in this module.

How about printing something a little more interesting than an arrow or an at sign on the screen. Suppose you wanted to print four lines in succession, which would look like a baton twirling or a pinwheel. Four lines which are available in the internal character set are listed below.

	HEX	DECIMAL
| =	$7C	124
/ =	$0F	15
− =	$0D	13
\ =	$3C	60

One possiblity is to repeatedly load the accumulator with the values for each of the four lines. For example, we could write the following program.

```
 *=$600
SCREEN = $9C40
 LDA #$7C         ;CODE FOR |
 STA SCREEN       ;DISPLAY
 LDA #$0F         ;CODE FOR /
 STA SCREEN       ;DISPLAY
 LDA #$0D         ;CODE FOR -
 STA SCREEN       ;DISPLAY
 LDA #$3C         ;CODE FOR \
 BRK              ;BREAK
```

It works, but this certainly is an inefficient way of displaying a pinwheel. Instead, it would be preferable to have one set of instructions that printed a line on the screen. The hexadecimal value for each of the different lines would be passed through the print routine in succession. This would eliminate the repetition of LDA and STA instructions. In assembly language it is possible to set up a data table, and read through the data, one element at a time, just the way you can in BASIC.

To store the codes for these lines as data in memory, the psuedo opcode ".BYTE" can be used. The .BYTE command informs the assembler that what follows is a series of bytes which are to be stored in successive memory locations. Not every assembler uses the .BYTE command. Some assemblers have other psuedo opcodes for saving data, such as HEX. To use the .BYTE command, the data must be listed in decimal and separated by commas. The .BYTE command that holds the data for the four lines is listed below.

```
Label   Psuedo Opcode Instruction
    /    \
  CHAR .BYTE  124,15,13,60
```

CHAR is the label used to identify where the data are stored in memory. The data are listed in the operand of the command field. Each number in the list of data must be equal to or less than 255, since each element of data is stored in one memory location. When the assembler converts the source code to object code, an address is assigned to the label CHAR. If the address of CHAR is $060E, then the first element of data following .BYTE will be stored in $060E. The second element of data will be stored in $060F and so on.

Address	Data
$060E	$7C
$060F	$0F
$0610	$0D
$0611	$3C

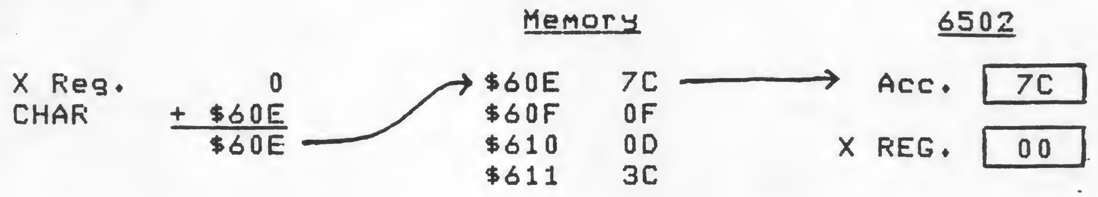
Now that the data are stored in memory, we need to be able to get the numbers to be printed on the screen, one at a time. Reading through data in assembly language is accomplished with "indexed addressing." The X register or the Y register serves as an "index" for reading through the data. The following format is used for indexed addressing.

```
LDA CHAR,X
```

The number in the X register is added to the address of CHAR. The value in this new address is loaded into the accumulator. For example, suppose the X register contains a zero.

```
LDA CHAR,X
    /    \
 $060E + 0 = $060E
```

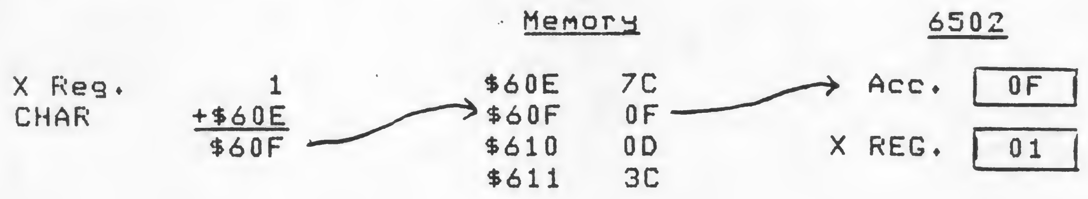
Zero is added to $060E, the address of CHAR. The accumulator is loaded with the contents of this new address.



A copy of the first byte of the CHAR data table is loaded into the accumulator. Suppose we incremented the X register to one.

```
        LDA CHAR,X
             /   \
     $060E + 1 = $060F
```

This time the value in $060F is loaded into the accumulator.



Either the X register or the Y register can be used as an index. With indexed addressing you can access any one of 255 elements of data stored in memory. You are restricted to a maximum index of 255, because that is the largest number the X or the Y register can hold. Turn to Assembly Language Programming Worksheet #12 to see how you can incorporate indexed addressing and the .BYTE psuedo opcode into your assembly language programs.

Assembly Language Programming Worksheet #12

1. ENTER and assemble the PINWHEEL program on the Advanced Topics Diskette. The listing on your screen should match the listing below. (The first line will not show since the screen scrolls up.)

```
0000          0100          *=$600          ;ORGIN
9C40          0110  SCREEN = $9C40          ;SCREEN RAM
0600 A200     0120          LDX #$00        ;SET INDEX TO 0
0602 BD0E06   0130  NEXTCHAR LDA CHAR,X     ;GET NEXT CHAR
0605 8D409C   0140          STA SCREEN      ;DISPLAY IT
0608 E8       0150          INX             ;ADD ONE TO INDEX
0609 E004     0160          CPX #$4         ;COMPARE X REG. TO 4
060B D0F5     0170          BNE NEXTCHAR    ;IF X <> 4 BRANCH
060D 00       0180          BRK             ;BREAK
060E 7C       0190  CHAR    .BYTE 124,15,13,60  ;DATA
060F 0F
0610 0D
0611 3C
```

2. Take a look at the object code.

3. What is the opcode for the LDA in the CHAR,X instruction?_______ Another opcode for the LDA instruction! "BD" instructs the processor to take the contents of the X register, add it to the address of CHAR, and store the contents of the new address in the accumulator. The opcode also tells the CPU to fetch two bytes in the operand following the opcode BD. The two bytes following the BD in the object code are the address of CHAR, where the first byte of the CHAR data table is stored.

4. Now look down at the contents of $060E - $0611. These are the bytes of data for the four lines that make the pinwheel. Note that there is no opcode for the .BYTE instruction. Psuedo opcodes are instructions to the assembler. They are not processed by the CPU. Also note that the .BYTE instruction and the pinwheel data are listed in the program following the BRK instruction. The data table must follow the BRK, because the data does not contain an instrucion or opcode for the CPU to execute. If the data came before the BRK, the CPU would try to interpret the data as opcodes to be executed.

5. A new instruction appears on line 160. "CPX" is one of a series of "compare" instructions.

CMP:	CoMPare Memory and the Accumulator
CPX:	ComPare Memory and the X Register
CPY:	ComPare Memory and the Y Register

The branch instructions we used earlier in this module branched until either 0 or 255 was reached. Compare instructions enable the programmer to devise a loop with a termination point other than 0 or 255. CPX compares the contents of the X register with the number in the operand of the compare instruction. CPX #$4 compares the contents of the X register with 4. The comparison is made by subtracting the operand, 4, from the value held in the X register. In the PINWHEEL program the X register is incremented just prior to the compare instruction. So the first time the CPX #4 is executed, the X register is one.

```
          CPX #$4

            01    X Register
           -04    Compare Operand
           ---
            -3
```

The answer, -3, sets the negative bit of the status register. Compare instructions set the negative, zero, or carry bit of the status register, depending on the results of the subtraction. There is no other evidence of the subtraction or execution of the compare instruction. The number in the X register remains the same as it was prior to the compare instruction. When the X register is incremented to four and compared to the 4 in the CPX instruction, the result of the comparison is zero.

```
          CPX #4

            04    X Register
           -04    Compare Operand
           ---
            00
```

The result of the comparison will set the zero flag of the status register. In the PINWHEEL program a BNE (branch not equal to zero) instruction is used to check the zero flag of the status register. Thus, the first through the fourth elements of data will be loaded into the accumulator and stored on the screen with indexed addressing. When the X register is incremented to 4, the BNE is no longer effective. The zero bit has been set, so the branch is not taken, and the next instruction in the program is executed.

6. Finally, let's run the program.

   Type: BUG   and press <RETURN>

   Type: G600   and press <RETURN>

According to the way the program was planned, you should see the four lines displayed, one right after the other, giving the appearance of one twirl of a baton. However, all you see is one line. We are up against a speed problem again. The computer is processing the program and displaying the lines so fast that all you can see is the last line. To be sure that each of the four lines is being printed, replace the BRK instruction at the end of the program with a jump back to the beginning of the program. Use the <CTRL> and arrow keys to place the cursor over the "B" in BRK.

Type: JMP BEGIN and press <RETURN>

The JMP instruction is similar to a GOTO in BASIC.

7. To insert the label BEGIN, place the cursor over the space before the LDX #$00 instruction. Hold down the <CTRL> key and press the <INSERT> key (in the upper right hand corner of the keyboard) five times - once for each letter in the word BEGIN.

Type: BEGIN and press <RETURN>

After you have typed BEGIN, be sure that there is a space in between the label BEGIN and the command LDX. Using the <CTRL> and arrow keys again, move the cursor down below the program.

8. Assemble the program and execute it from $600. At least we now know that each of the four lines is being stored in screen RAM as we intended.

To make the pinwheel look more like it is spinning, we need a brief delay after displaying each line. Ideally, we would simply insert a JSR DELAY into the routine that draws the pinwheel. However, we must first review how each of the subroutines is using the registers. It may be that one subroutine changes a register and affects the operation of the second routine. Look over the listing below. Focus on the use of the X register.

```
 *=$600          ;ORIGIN
SCREEN = $9C40 ;SCREEN RAM
;
DRAW LDX #$00  ;SET INDEX TO 0
NEXTCHAR LDA,X ;GET NEXT CHAR
 STA SCREEN    ;DISPLAY IT
 JSR DELAY     ;CALL DELAY ROUTINE
 INX           ;ADD 1 TO INDEX
 CPX #$4       ;COMPARE X REG. TO 4
 BNE NEXTCHAR  ;IF X=4 THEN BRANCH FOR CHAR
 BRK           ;BREAK
CHAR .BYTE 124,15,13,60  ;PINWHEEL DATA
;
;
DELAY LDX #$A0 ;COUNTER FOR Y LOOPS
AGAIN LDY #$00 ;BEGIN WITH 0
WAIT INY       ;ADD 1 TO Y
 BNE WAIT      ;IF NOT 0, REPEAT WAIT
 DEX           ;SUBTRACT 1 FROM X
 BNE AGAIN     ;IF NOT 0, REPEAT AGAIN
 RTS           ;RETURN FROM SUBROUTINE
```

The X register is used both as an index to CHAR, and as a counter in the DELAY loop. The DRAW routine sets the X register to zero and loads the accumulator with the character to be printed on the screen. Then a delay is needed, so we JSR DELAY. In the course of the DELAY loop, both the X and the Y registers are manipulated. However, they are both at zero when the subroutine is completed. Thus, there is no conflict in the use of the X register the first time through the program. However, the Draw routine increments the X register in order to read through the line data. Suppose the X register has been incremented to one. When the DELAY loop is called, the X register is reset to zero. Immediately following the DELAY routine, the DRAW routine increments X. Consequently, the index to the data will be continuously reset to zero by DELAY and incremented to one in the DRAW routine. Since the X register would never get to four, the program would branch continuously to NEXTCHAR, and display the same data line over and over again. Thus, we need some way to preserve the index that reads through the data.

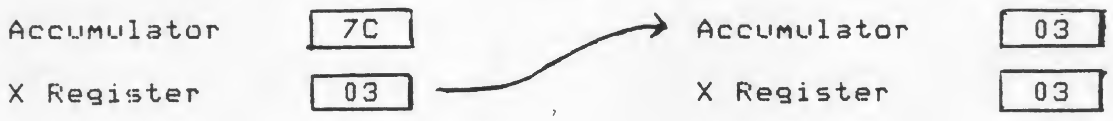
This is a good opportunity to employ the "stack," an area of memory reserved for temporary storage of information. Before calling the DELAY routine, we will save the current value of the index on the stack.

In the Machine Architecture module the "PHA" and "PLA" instructions were introduced. PHA stands for PusH the Accumulator onto the stack. PLA, PuLl the Accumulator off the stack, is used to retrieve the value from the stack. Any value to be put on the stack must first be put in the accumulator. So in order to save the X register on the stack, first we need to put the value in the X register into the accumulator. To shift a value from one register to another, we need to use one of a set of "transfer" instructions.

TXA:	Transfer the contents of the X register to the Accumulator.
TAX:	Transfer the contents of the Accumulator to the X register.
TYA:	Transfer the contents of the Y register to the Accumulator.
TAY:	Transfer the contents of the Accumulator to the Y register.

Transfer instructions store a copy of the value in one register in another register, as shown below.

TXA



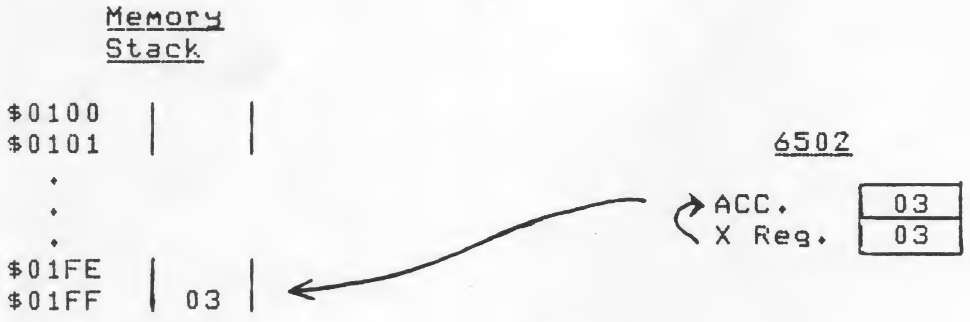
A copy of the X register is put in the accumulator. The X register remains intact.

None of the transfer instructions require an operand. All of the information the CPU needs is evident from the instruction, so implied addressing is used. Glance over the use of the PHA, PLA, and the transfer instructions below.

```
        TXA               ;TRANSFER X INDEX TO ACCUMULATOR
        PHA               ;SAVE IT ON THE STACK
        JSR DELAY         ;CALL DELAY ROUTINE
        PLA               ;RETRIEVE INDEX FROM STACK TO ACCUMULATOR
        TAX               ;TRANSFER INDEX FROM ACCUMULATOR TO X
                          ;REGISTER
```

The index in the X register is transferred to the accumulator. PHA pushes the index, which is now in the accumulator, onto the stack. (The stack fills from $01FF down to $0100.)

1. TXA
2. PHA



The JSR DELAY sends the CPU to the address of DELAY to execute the subroutine. When the DELAY loop is completed, it returns the CPU to the instruction following the JSR DELAY in the DRAW routine. PLA retrieves the index from the stack and puts it into the accumulator. TAX transfers the index, in the accumulator, back to the X register. Turn to Assembly Language Programming Worksheet #13 to see how this sequence of instructions has been incorporated into the DRAW routine. This time the pinwheel will spin.

Assembly Language Programming Worksheet #13

1. ENTER the SPIN program on your Advanced Topics Diskette.

    Type: ENTER #D:SPIN

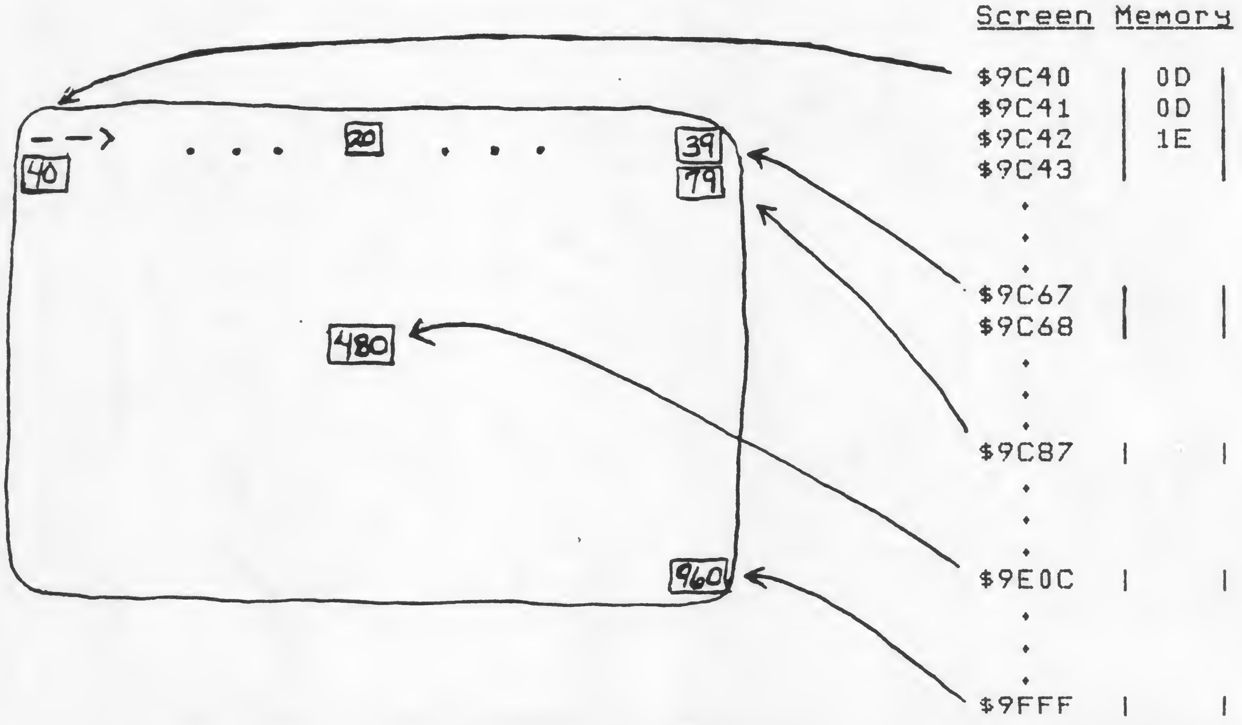
2. LIST lines 150 to 250 to see how the transfer commands have been incorporated into the DRAW routine. A complete listing of the program appears at the back of this module.

3. Assemble the program and execute it from the debugger at $600.

4. You can transfer the accumulator to the X register, and the X register to the accumulator. The Y register also can be transferred to the accumulator and vice versa. However, there is no instruction for transferring data between the X and Y registers. How can you transfer the X register to the Y register using the transfer commands you have learned? Write the assembly langauge code below.

Command	Comments
--------------------	----------------
--------------------	----------------
--------------------	----------------
--------------------	----------------

Spinning the pinwheel in the corner of the screen is fun, but how about putting that pinwheel somewhere else on the screen? The graphics zero screen has 960 locations, and so there are 960 memory locations reserved, each of which correspond to one location on the screen. Up until now, we have been using $9C40, the "starting location" of the graphics zero screen. There are 40 locations per line and 24 lines on the graphics zero screen. If you multiply 40 by 24, you come up with the 960 locations on the screen mentioned earlier. The 40 locations on the top row of the screen are numbered from 0 to 39 in decimal, and correspond to memory locations $9C40 - $9C67. The second row is numbered 40-79. The corresponding addresses are $9C68 - $9C8F. The address of the middle of the screen is $9E0C, and the contents of the last location on the graphics zero screen is stored at $9FFF.

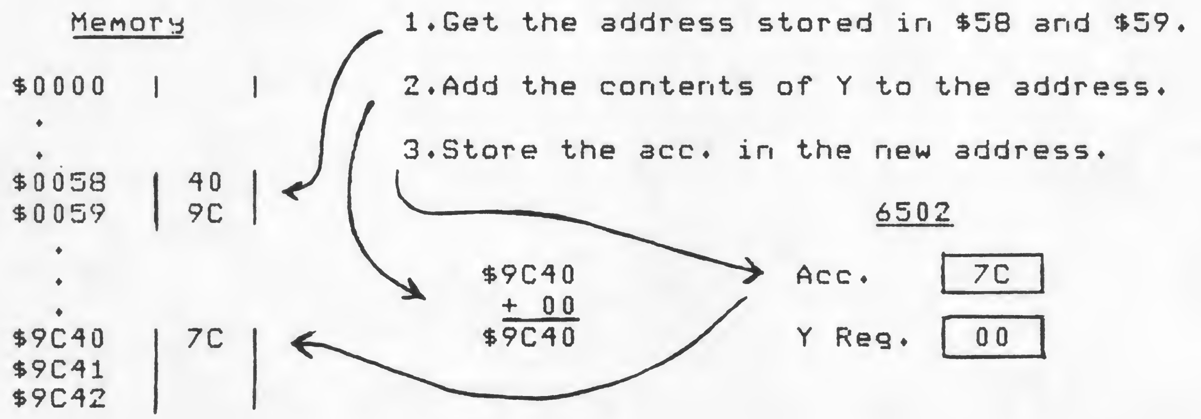


In order to move the pinwheel around on the screen, we need to be able to access any one of the 960 addresses ($9C40 - $9FFF) in screen RAM. One solution is to use "indirect indexed addressing." Indirect indexed addressing requires that the address to be indexed is stored on the zero page of memory. Quite conveniently, the starting address of screen RAM is stored in $58 and $59 on the zero page. Ordinarily, memory locations $58 and $59 hold $9C40 which is the default starting address for the screen. See the Internal Representation of Graphics and Text module for an explanation of how the different graphics modes use memory. For our present purposes we will use the $9C40 stored in $58 and $59 on the zero page. The low order byte of the address, 40, is stored in $58. The high order byte of the address is stored in $59.

Indirect indexed addressing uses the Y register as an index. An example of an indirect indexed instruction is listed below.

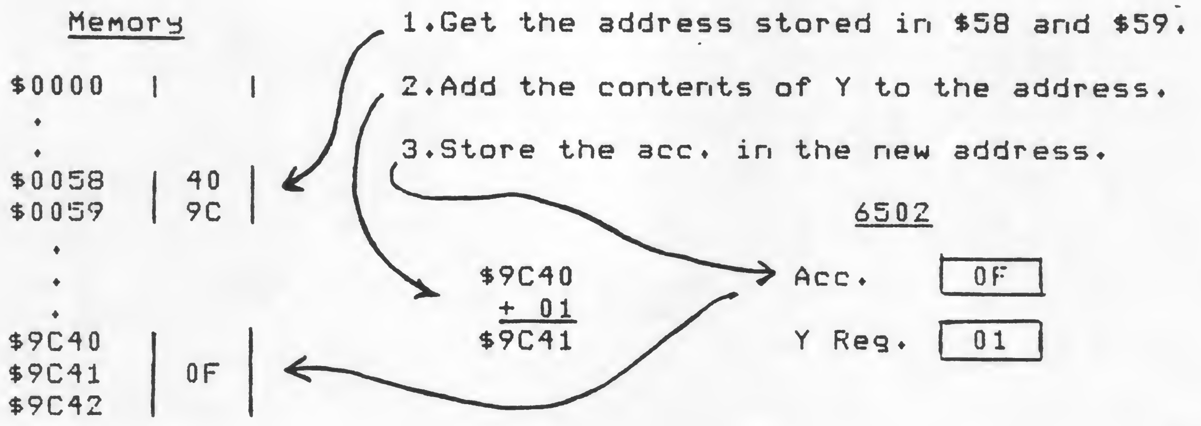
```
STA ($58),Y
```

When the CPU encounters an opcode for indirect indexed addressing, it automatically takes the low byte of the zero page address given in the instruction and looks for the high order byte of the address in the next memory location. Thus, the CPU gets the address contained in $58 and $59. Then the value in the Y register is added to the address. The STA instruction stores the value in the accumulator into the new address. Look over the diagram of the STA ($58),Y command below.



Memory
$0000
$0058 40
$0059 9C
$9C40 7C
$9C41
$9C42
1.Get the address stored in $58 and $59.
2.Add the contents of Y to the address.
3.Store the acc. in the new address.
$9C40
+ 00
$9C40
6502
Acc. 7C
Y Reg. 00

The STA instruction stores the accumulator in $9C40. Suppose the value in the Y register were incremented to one. To execute the STA ($58),Y instruction, first the CPU fetches the address stored in $58 and $59. In our example the address is $9C40. Then one, from the Y register, is added to the address. The STA instruction uses this final address to store the value in the accumulator in memory. Look over the diagram below.


Memory
$0000
$0058
40
$0059
9C
$9C40
$9C41
0F
$9C42
1.Get the address stored in $58 and $59.
2.Add the contents of Y to the address.
3.Store the acc. in the new address.
6502
Acc.
0F
Y Reg.
01
$9C40
+ 01
$9C41

The address stored in $58 and $59 has not been changed. In the programs that follow, the names LOWSCR and HISCR have been assigned to $58 and $59, because they hold the low byte and the high byte of screen RAM.

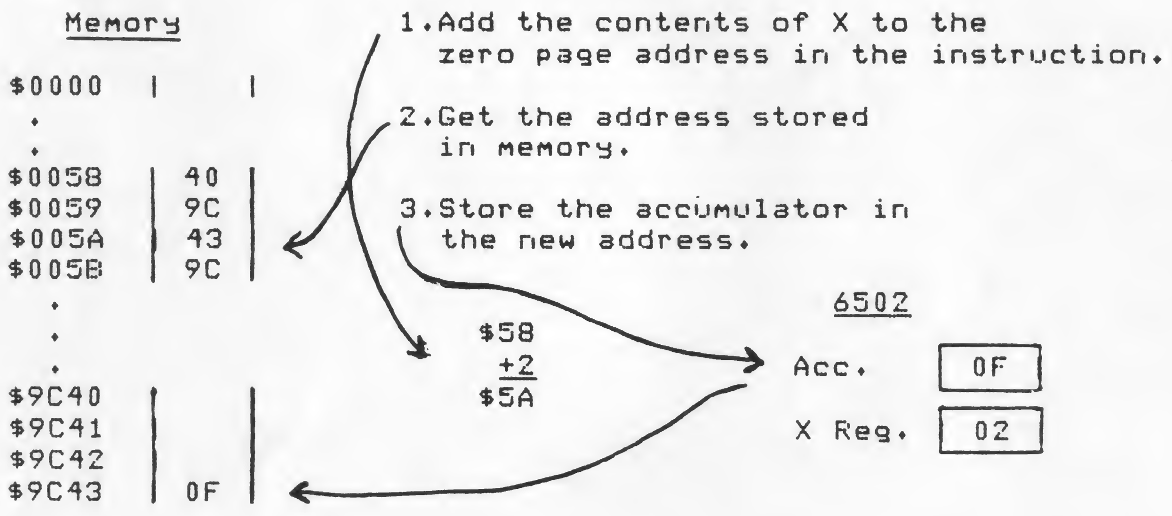
This is fairly difficult to understand at first. Don't panic. As you start programming in assembly language, you will see more applications for indirect indexed addressing, and it will become easier to understand.

There is one remaining 6502 addressing mode, which will not be used in the final animation program. "Indexed indirect" addressing is the least commonly used addressing mode in assembly language. Only the X register can be used as an index in indexed indirect addressing. An instruction using indexed indirect addressing looks like the this:

```
STA ($58,X)
```

The value in the X register is added to the zero page address in parentheses. This new address contains another address. The accumulator is stored in this last address. Suppose the X register holds a 2 and the CPU is executing a STA ($58,X) instruction.

STA ($58),X


Memory
$0000
$0058 40
$0059 9C
$005A 43
$005B 9C
$9C40
$9C41
$9C42
$9C43 0F
1.Add the contents of X to the zero page address in the instruction.
2.Get the address stored in memory.
3.Store the accumulator in the new address.
$58
+2
$5A
6502
Acc. 0F
X Reg. 02

Thus, the value in the X register is added to the zero page address in order to get another memory address which is stored on the zero page. Indexed indirect addressing is useful when you wish to access a certain element of data from various equal sized data tables stored in memory. You needn't worry if you don't understand the indexed indirect addressing mode just yet.

Challenges

A lot of material has been cover. This is a nice opportunity for you to experiment with what you have learned. Select one of the challenges listed below. Instructions for loading, saving, and printing assembly language programs are provided in a reference list at the back of this module.

1. Print a message on the screen using indexed addressing. The characters in the message should be stored in a data table using the .BYTE psuedo opcode. Use indexed addressing to access the characters in the message one at a time. Also use indexed addressing to increment your screen RAM locatons for the output.

2. Look through the internal character set chart at the back of this module. Select a series of characters to be displayed in one location on the screen in a sequence to suggest animation. Store the characters you have selected in a data table. Use indexed addressing to access the characters one at a time. Animate them at the center of the screen. You will need to use the stack to preserve the X register index used for indexed addressing, since the X register will be needed for a DELAY routine as well.

3. Write a program to move a "greater than" sign (>) across the screen. Assign a label to the starting screen address at the beginning of your program (eg. SCREEN = $9C40). Use indexed addressing with the label SCREEN to move the symbol across the video monitor. Don't forget to call a DELAY routine before displaying the symbol in successive SCREEN locations.

Animation

In this section you will write the assembly language routines necessary to move the pinwheel around on the screen. You also will learn how to read joystick input and move the pinwheel in the direction the joystick has been pushed.

First let's start by moving the pinwheel to the right across the screen. To move the pinwheel to the right, we need to add one to the pinwheel's current address in screen RAM. The address of screen RAM on the zero page will be continually updated as the pinwheel is moved. We will still use indirect indexed addressing. But, instead of incrementing the Y register, we will add one to the screen RAM address of the pinwheel's current position.

Adding is done with the "ADC" instruction, which stands for ADd with Carry. ADC adds the accumulator and the carry bit of the status register to the operand of the ADC instruction. The sum is stored in the accumulator. ADC #$1, adds one to the value in the accumulator, plus the carry bit. The example below illustrates the possible results of an ADC instruction. The sum of the addition is always stored back in the accumulator.

```
ADC #$1          $40  Accumulator          $40  Accumulator
                  01  Add Operand           01  Add Operand
                   0  Carry Bit Clear        1  Carry Bit Set
                 $41                       $42
```

The result of the addition will be $41 or $42, depending on whether the carry bit is set or not. Before adding, you need to clear the carry bit, unless you want to include the carry in an addition. Clearing the carry bit will insure the accuracy of your addition. The "CLC" instruction is used to CLear the Carry flag of the status register. CLC uses implied addressing. No operand is needed. An assembly language routine which adds one to the address of screen RAM is listed below.

```
CLC             ;CLEAR THE CARRY BIT TO 0
LDA LOWSCR      ;LOAD THE ACC. WITH THE LOW BYTE OF SCREEN RAM
ADC #$1         ;ADD 1 TO THE ACCUMULATOR
STA LOWSCR      ;STORE THE ACC. IN THE LOW BYTE OF SCREEN RAM
LDA HISCR       ;LOAD ACC. WITH THE HIGH BYTE OF SCREEN RAM
ADC #$00        ;ADD ZERO TO THE ACCUMULATOR
STA HISCR       ;STORE THE SUM IN HISCR
BRK             ;BREAK
```

Does it seem strange that one is added to LOWSCR and then zero is added to HISCR? Imagine the situation where LOWSCR is $FF and HISCR is $9C ($9CFF). Now add one to LOWSCR.

```
                              $ FF   LOWSCR
                              $ 01   ADC #$1
           Carry Bit = 1      $ 00   Accumulator
```

The answer in the accumulator will be zero and the carry bit is set. The new screen RAM address is $9C00. The high byte of the address,(9C), remains the same. However, $9C00 does not follow $9CFF in screen RAM -- $9D00 does. The carry bit needs to be added to the high order byte of the screen address. That explains the addition with HISCR. The carry bit was cleared before adding one to the low order byte of the address. If the carry was set by the first addition, a one will be included in the addition when zero is added to the high order byte of the address.

```
      LDA LOWSCR                  LDA HISCR
      ADC #$1                     ADC #$00


           $ FF  LOWSCR         $9C  HISCR
              1  ADC #$1         00  ADC #$00
              0  CLC              1  Carry Set by LOWSCR Add.
Carry = 1  $ 00                 $9D
                 \             /
                      $9D00
```

If the carry bit is not set by the first addition, zero is added to the high byte of the address, and it goes unchanged.

```
      LDA LOWSCR                  LDA HISCR
      ADC #$1                     ADC #$00


           $ 40  LOWSCR         $9C HISCR
              1  ADC #$1         00  ADC #$00
              0  CLC              0  Carry
Carry = 1  $ 41                 $9C
                 \             /
                      $9C41
```

Turn to Assembly Language Programming Worksheet #14 to see how this addition routine can be incorporated into the program to make the pinwheel move to the right across the screen.

Assembly Language Programming Worksheet #14

1. ENTER the ANIRIGHT program on your Advanced Topics Diskette.

As your programs get longer and more complex, it becomes necessary to set up a "main loop," which "calls" each of the subroutines.

2. To see the main loop in the ANIRIGHT program, list lines 120-180 or look at the listing of the ANIRIGHT program at the back of this module.

Type: LIST 120,180 and press <RETURN>

You will notice a list of JSR's to different subroutines in the program. The main loop listed below has been inserted into the beginning of the program, following the constant and variable declarations.

```
BEGIN JSR DRAW       ;JUMP TO THE PINWHEEL DRAW
      JSR DELAY      ;PAUSE WHILE DISPLAY PINWHEEL
      JSR RIGHT      ;MOVE THE PINWHEEL TO THE RIGHT
      JMP BEGIN      ;JUMP BACK TO BEGIN AND
                     ;RE-EXECUTE THE LOOP
```

The first JSR DRAW draws the pinwheel in its starting position. The JSR DELAY holds the pinwheel in place momentarily, so we can see it before it is moved to the right. JSR RIGHT calls the routine that adds one to the address of the pinwheel's position on the screen. In order to see the pinwheel move, we want to draw the pinwheel again in its new position. Instead of adding another JSR DRAW, the next instruction, JMP BEGIN, sends the CPU back to the label BEGIN, and the first JSR DRAW is re-executed. The screen address has been updated, so the pinwheel is drawn in its new location.

3. LIST 450-550 and you will see that the addition routine has been incorporated into the program.

Type: LIST 450,550 and press <RETURN>

4. Don't forget that by using indirect indexed addressing to display lines on the screen, we have added another use of the Y register to the program. However, both the DRAW routine and the DELAY routine reset the Y register to zero. Thus, the additional use of the Y register does not effect the subroutines.

5. Assemble and execute the program from the debugger.

The main loop in this program is an infinite loop. To stop the program you must press <SYSTEM RESET>. If you let the ANIRIGHT program continue past the last location in screen memory, the program will continue to store the code for the pinwheel in successive memory locations. The last address of screen RAM is $9FFF. The assembler editor is stored in memory starting at $A000. If you let the ANIRIGHT program continue, you may write over the assembler editor in memory with pinwheel data. If this occurs, the EDIT prompt will not come on the screen when you press <SYSTEM RESET>. In that case, you will have to reboot the system.

6. Why are all those extra lines left on the screen?

Animating shapes in BASIC and assembly language requires the same sequence of steps.

1. Set up the location for the pinwheel on the screen.
2. Draw the shape.
3. Hold the shape on the screen with a delay.
4. Erase the shape.
5. Repeat the cycle.

The cycle is continued as long as the shape is being animated.

In the ANIRIGHT program, we need an erase routine to draw over the last line of the pinwheel, before a pinwheel is drawn in the next position on the screen. To erase the line, store a space in the pinwheel's most recent screen position. Look over the ERASE routine listed below.

```
ERASE LDY #$00        ;INDEX FOR ZERO PAGE ADDRESSING
      LDA #$00        ;CHARACTER CODE FOR SPACE
      STA (LOWSCR),Y  ;STORE OVER LAST PINWHEEL
      BRK             ;BREAK
```

The ERASE routine is really quite simple. Indexed indirect addressing is used to store the space in the pinwheel's most recent position. Turn to Assembly Language Worksheet #15 to see how the RIGHT program has been changed by incorporating the ERASE routine.

Assembly Language Programming Worksheet #15

1. ENTER the program called ERASE on the Advanced Topics Diskette.

2. LIST lines 550-650 to see that the ERASE routine has been added. The ERASE routine is called from the main loop.

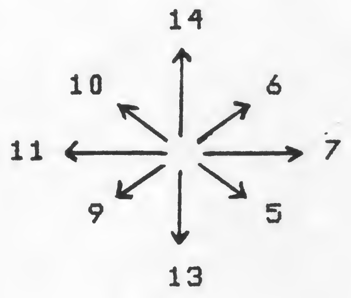
    Type: LIST 550,650 and press <RETURN>

An entire listing of the ERASE program appears at the back of the module.

3. Assemble the program and run it from the debugger. Remember to press <SYSTEM RESET> to get back to the EDIT prompt. Otherwise, you will have to reboot the system.

4. When the pinwheel reaches the right edge of the screen, it comes back on the left side of the screen, one line down. What do you think causes the pinwheel to "wrap around" the screen?

Now add joystick control. To move the pinwheel with the joystick, you must first know which direction the joystick is being pushed. Values are assigned to the different positions of the joystick.



When the joystick is pushed to the right, the number 7 is stored in a memory location reserved for joystick input. Which memory location holds the 7 depends on which "port" (on the front of the Atari) the joystick is plugged into. If the joystick is plugged into the first port on the far left, the 7 will be stored in memory location $278 (632 in decimal). To see which direction joystick #1 has been pushed, you simply read the contents of $278. The memory addresses reserved for joystick input are listed below.

Joystick in Port #1	$278
Joystick in Port #2	$279
Joystick in Port #3	$27A
Joystick in Port #4	$27B

One way to read the contents of a memory location is to load the accumulator with the value and do a series of comparisons. For example, LDA $278 loads the accumulator with the most recently depressed direction of joystick #1. To check the value we can compare the accumulator with the specific values we are looking for. If we compare the contents of the accumulator with 7 and find that the value is 7, we know that the joystick has been pressed to the right. An assembly language routine that compares the joystick reading with the values for left and right is listed below.

```
LDA #$278       ;READ JOYSTICK PORT #1
CMP #$7         ;IS IT A 7?
BEQ RIGHT       ;IF SO, BRANCH TO THE RIGHT ROUTINE
CMP #$B         ;IS IT 11?
BEQ LEFT        ;IF SO, BRANCH TO THE LEFT ROUTINE
```

Comparisons are only made with those values for the directions we are looking for. Any other value returned from the joystick in $278 is ignored. Thus, if the joystick is pressed on a diagonal, a 6 will be loaded into the accumulator. When the comparisons are made for a left or a right joystick press, the 6 will be ignored since the 6 does not match the 7 for right, or the 11 for left.

RIGHT and LEFT are labels for subroutines which change the pinwheel's direction of travel. Turn to Assembly Language Worksheet #15 to see how the joystick reading routine can be incorporated into the program.

Assembly Language Programming Worksheet #15

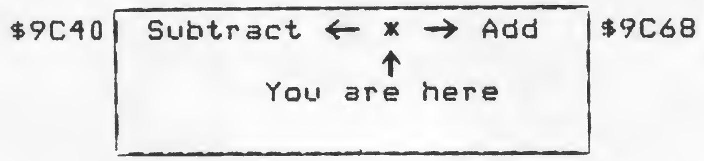
1. ENTER the JOYMOVE program on your Advanced Topics Diskette.

2. LIST lines 150-220 or look over the listing of the JOYMOVE program at the back of this module.

A JSR JOYSTICK command has been added to the main loop. The JOYSTICK routine gets directional feedback from the joystick. Whenever the person using the program pushes the joystick to the right, the RIGHT routine is called from the JOYSTICK routine.

3. LIST lines 100-150 to see how the name "STICK" has been assigned the address $278 in the constant declarations at the top of the program. For anyone reading through the program, the name STICK is much easier to understand than the hexadecimal address $278.

4. You have a routine to move the pinwheel to the right. Now you need a routine to move the pinwheel to the left. Since the address of each position on the screen is numbered from left to right, instead of adding, you need to subtract one from the screen address in order to move the pinwheel to the left.



When we wrote the add routine, first we had to clear the carry bit of the status register with the CLC instruction. The opposite is true for subtraction. Before subtracting you need to SEt the Carry bit with an "SEC" instruction. This is due to a peculiarity of the CPU's numbering system. If you would like an explanation of why you must set the carry bit before subtracting, see Chapter 9 of The Atari Assembler, by Don and Kurt Inman. There are copies in the camp library.

The format of the subtraction subroutine is identical to the addition routine. The carry bit is set with the SEC instruction. The "SBC", SuBtract with Carry instruction, subtracts the number in its operand and the carry bit from the accumulator. The result is stored back in the accumulator. "Double precision" arithmetic, where the high byte of an address must be updated based on the results of the low byte arithmetic, is repeated in this routine. Try writing your own routine which moves the pinwheel to the left.

5. LIST lines 300-380 to review the RIGHT routine. Now try writing a left routine below.

```
     LEFT _____SEC_______;SET THE CARRY BIT

          _______________;LOAD THE ACC. WITH LOWSCR

          _______________;SUBTRACT $1 FROM THE ACCUMLATOR

          _______________;STORE THE ANSWER IN LOWSCR

          _______________;LOAD THE ACCUMULATOR WITH HISCR

          _______________;SUBTRACT ZERO FROM VALUE IN ACC.

          _______________;STORE THE ANSWER IN HISCR

          _______________;RETURN (or BRK when typed in as
                           a separate routine)
```

LIST lines 390-460, to compare your subroutine with the LEFT routine in the JOYMOVE program.

6. Assemble the program and run it from the debugger. You should be able to move the pinwheel to the right or left with the joystick. Since there is no UP or DOWN routine, the pinwheel will not respond when you press the joystick in those directions. The program is in a continuous loop, which reads the joystick and moves the pinwheel continuously. You must press <SYSTEM RESET> to stop the program. You will be returned to the editor. How can you change the program so that it is not an infinite loop?

____

____

____

Assembly Language Programming Worksheet #16

Now all you need are two routines that move the pinwheel up and down.

1. The subroutine that moves the pinwheel down one line is identical to the RIGHT routine, except for the number that is added to the LOWSCR address. If there are forty spaces per line, how much should be added to the LOWSCR address to move the pinwheel down one row?_______

2. LIST lines 300-380 of the JOYMOVE program to review the RIGHT routine. Try writing your own DOWN routine. Fill in the blanks below.

```
DOWN  ______________;CLEAR THE CARRY

      ______________;LOAD THE ACCUMULATOR WITH LOWSCR

      ______________;ADD 40 TO ACC. PLUS CARRY

      __STA_LOWSCR__; _________________________

      ______________;LOAD THE ACCUMULATOR WITH HISCR

      ___ADC_#$00___; _________________________

      ______________;STORE THE ACCUMULATOR IN HISCR

      ______________;RETURN (or BREAK when typed in as
                      a separate routine.)
```

3. Now write a routine that will move the pinwheel UP the screen.

```
UP  _____SEC______;SET THE CARRY BIT

    ______________;LOAD THE ACCUMULATOR WITH LOWSCR

    ______________;SUBTRACT 40 FROM THE ACCUMULATOR

    ______________;STORE THE ACCUMULATOR IN LOWSCR

    ______________;LOAD THE ACCUMULATOR WITH HISCR

    ______________;ADD ZERO AND THE CARRY BIT TO HISCR

    ______________;STORE THE ACCUMULATOR IN HISCR

    ______________;RETURN (or BREAK if typed in as
                   a separate routine.)
```

4. The last set of instructions that need to be updated before the animation is complete, is the joystick routine. The UP and DOWN routines need to be included in the JOYSTICK reading routine. The current listing of the JOYSTICK routine is printed below. Complete the comparisons and branches to the UP and DOWN routines.

```
JOYSTICK LDA STICK   ;LOAD ACC. WITH JOYSTICK PRESS
         CMP #$7     ;COMPARE THE JOYSTICK INPUT TO 7 - RIGHT
         BEQ RIGHT   ;IF EQUAL TO 7 THEN BRANCH TO RIGHT
         CMP #$B     ;TO THE LEFT?
         BEQ LEFT    ;IF SO BRANCH TO LEFT ROUTINE

         ______ #$D  ;COMPARE JOYSTICK INPUT TO 14 FOR UP

         ______ UP   ;IF EQUAL, THEN BRANCH TO UP

         ______ #$C  ;COMPARE JOYSTICK INPUT TO 13 FOR DOWN

         ______ DOWN ;IF EQUAL TO 13 THEN BRANCH TO DOWN

         ___________ ;RETURN
```

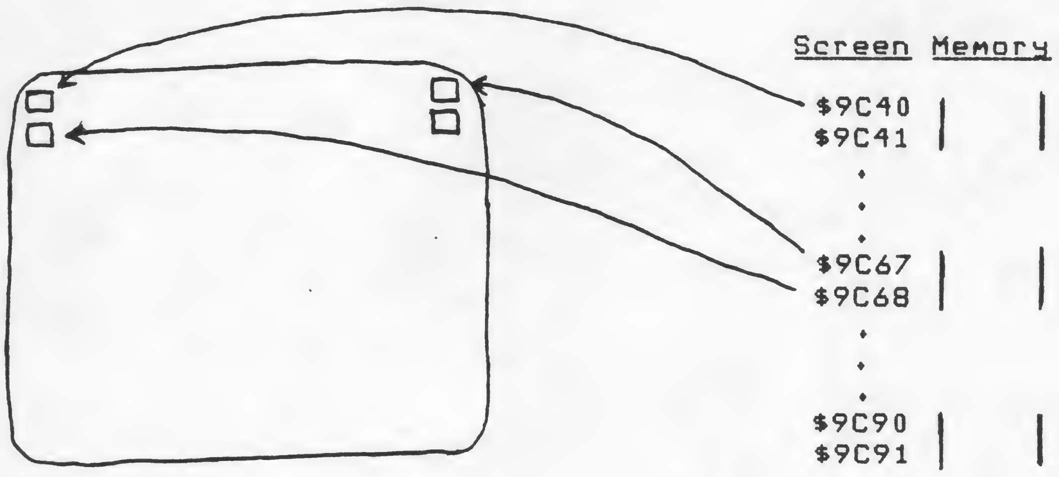
5. ENTER the ANIMATE program on your Advanced Topics Diskette.

6. LIST lines 510-660. Compare your DOWN and UP routines with the ones in the ANIMATE program.

7. LIST lines 250-350 to check your JOYSTICK routine against the one in the ANIMATE program.

8. Now assemble the program and try it out.

The pinwheel moves in each of the four directions. When you move the joystick left or right and the pinwheel goes off the screen, it comes back on the screen on the opposite side. This is because screen memory is sequential from one row to the next. The address of the rightmost position on the top row of the screen is one less than the address of the leftmost position on the second row on the screen.



When you move the joystick up or down off the screen, peculiar things happen on the screen. This is because the pinwheel has moved out of screen RAM and is storing pinwheel data in areas of memory being used for other purposes. The program never checks where the pinwheel is in memory, it just adds or subtracts 40 from the pinwheel's position. Remember, all that exists in memory is a long string of boxes, each holding one number. It is the sequence of the numbers, and the CPU's interpretation of those numbers, that enables the computer to operate. If we store the values for the pinwheel and then erase the pinwheel in memory locations outside of screen RAM, we are leaving zeros in areas of memory that might have held important data or instructions for the CPU. Thus, when you move the pinwheel up or down off the screen, you may be writing over the data in memory, which is there for other purposes, and you may confuse the computer so much that <SYSTEM RESET> will not return you to the EDIT prompt. Instead, you will have to reboot the system.

In conclusion, we have set aside areas of memory to serve different functions. The zero page holds the screen RAM address, which we access with indirect indexed addressing. Memory locations $600-$686 hold our program. We are using memory locations $9C40-$9FFF to hold the data to be displayed on the screen. While the numbers in these memory locations bear significance to us, the programmers, and to the CPU, to someone who is unfamiliar with computers or assembly language, memory contains just a long, LONG, list of unintelligible numbers.

```
Use of Memory                                          Contents of Memory

$0000 |      |\                                        $0000 | 00 |
  .           |                                          .
  .           |                                          .
  .           |                                          .
$0058 |LOWSCR| > Zero Page                             $0058 | 40 |
$0059 |HISCR |                                         $0059 | 9C |
  .           |                                          .
  .           |                                          .
  .          /                                           .
$0600 | JSR  |\                                        $0600 | 20 |
$0601 | JOY- | |                                       $0601 | 0F |
$0602 |STICK | |                                       $0602 | 06 |
  .            |                                         .
  .            |                                         .
  .            |                                         .
$0650 | ADC  | >ANIMATE                                 $0650 | 69 |
$0651 |  28  | |Program                                 $0651 | 28 |
  .            |                                         .
  .            |                                         .
  .            |                                         .
$0683 | DEX  | |                                       $0683 | CA |
$0684 | BNE  | |                                       $0684 | D0 |
$0685 |OFFSET| |                                       $0685 | F8 |
  .            |                                         .
  .           /                                          .
  .                                                      .
$9C40 |  7C  |\                                        $9C40 | 7C |
$9C41 |      | |                                       $9C41 |    |
$9C42 |      | |                                       $9C42 |    |
  .            |                                         .
  .            >Screen RAM                               .
  .            |                                         .
$9FFE |      | |                                       $9CFE |    |
$9FFF |      | |                                       $9CFF |    |
              /
```

The USR Function

Suppose you want to write a program and you would like to have the benefit of the fast, smooth animation of assembly language, and you would also like to have the ease and convenience of BASIC PRINT statements for your user prompts. It may be that you would like to explain how to operate your animation program in BASIC before running the assembly language program. BASIC allows you to load and run an assembly language routine from a BASIC program with the USR function. You saw an example of this in the Machine Architecture module. Two programs were used to demonstrate the difference in performance between a BASIC program and an assembly language routine. The programs both filled the graphics zero screen with the character of the most recent keypress. The programs were called SCRFULL and FILLSCR. To refresh your memory, run the programs on the Advanced Topics Diskette. Both programs are in BASIC. Be sure that you have a BASIC cartridge in the computer. At the BASIC READY prompt,

Type: RUN "D:SCRFULL" and press <RETURN>

Press <SYSTEM RESET> to exit the program.

Now for comparison, run the BASIC program which POKES the data for an assembly language routine into memory and then executes the assembly language program to fill the screen.

Type: RUN "D:FILLSCR"

The difference in the performance of the two programs is significant.

The USR function is what enables you to run an assembly language routine from BASIC. First, however, the object code for the routine must be in memory. In the FILLSCR program, the assembly language routine is POKED into memory from BASIC. A listing of the FILLSCR program appears at the back of this module. The decimal equivalents for each byte of the object code are listed in DATA statements. A FOR .. NEXT loop is used to READ each byte of DATA and POKE it into memory. The the USR function is used to "call" the assembly language routine. The format of the USR function is as follows.

```
CALL = USR (starting address, parameter)

CALL = USR (1536,CHARACTER)
```

The number 1536 is the decimal equivalent to $600, which is the starting location of the assembly language routine in memory. Remember, decimal numbers must be used in BASIC. The variable CHARACTER, which follows the starting address of the routine in the USR call, is optional. Any values which are to be passed to the assembly language routine are listed after the starting address and separated by commas. In this example the keypress CHARACTER is being passed to the assembly language routine.

When BASIC executes the USR function, the 6502's registers and the program counter are stored on the stack until the assembly routine is completed. Any variables which are passed to the assembly routine from BASIC are treated as two byte values and also are put on the stack when the USR function is executed. Thus, the value for the CHARACTER will be stored on the stack in two successive memory locations as a two byte value. After each of the variables has been placed on the stack, BASIC puts one number on the stack, which indicates how many variables have been passed. In this case a one will be put on the stack.

Study the FILLSCR program, and then try writing a BASIC program to introduce the animation program. Store the object code for the animation routine in DATA statements and use a FOR ... NEXT loop to POKE the routine into memory. Call the routine with the USR function. For more information about how the USR function operates, see Chapter Three of The Atari Assembler by Inman and Inman in the camp library.

Summary

6502 Addressing Modes

Immediate: LDA #$50

Load the accumulator with the immediate value in the operand, $50.

Absolute: LDA $278

Load the accumulator with the contents of memory location $278. In absolute addressing, the operand is a 16 bit address or a label.

Zero Page: LDA $58

Load the accumulator with the contents of the zero page location $58. The low byte of a zero page address is listed in the operand.

Implied: CLC

Clear the carry bit. The operation to be performed is implied by the instruction. No operand is necessary.

Relative: BNE WAIT

Branch to WAIT as long as the zero bit of the status register is not set. Branches are made relative to the instructions being branched to. The CPU will not let you branch further than 127 bytes. Branch instructions are the only instructions that use relative addressing.

Indexed: LDA SCREEN,X

Add the X register to the SCREEN address. Load the accumulator with the contents of the new address. Either the X or the Y register can be used with indexed addressing.

Indirect Indexed: LDA($58),Y

Get the address stored in $58 and $59 on the zero page of memory. Add the Y register to the address. Load the accumulator with the contents of the new address. Indirect indexed addressing also is referred to as post indexed addressing.

Indexed Indirect: LDA ($58,X)

Add the X register to $58. If X is 2, get the address stored at $5A and $5B on the zero page. Load the accumulator with the contents of the address. This type of addressing is also called Pre-indexed addressing.

A chart of the 6502 instruction set and the corresponding opcodes is provided at the back of the module.

The appendices of The Atari Assembler, by Don and Kurt Inman, and 6502 Assembly Language Programming by Lance Leventhal, include detailed descriptions of the 6502 instruction set, addressing modes, and the status flags affected by each instruction. You can find copies of these books in the camp library.

Challenges

1. Write an assembly language program that prints your name in the middle of the screen. Use the .BYTE psuedo opcode and indexed addressing to print your name.

2. In the animation program, we are continually changing the position of the pinwheel stored in memory locations $58 and $59. Locations $58 and $59 are the locations the computer uses to hold the starting address of screen RAM. When a break occurs in the animation program, the computer uses the address it finds in $58 and $59 for the starting location on the screen. Consequently, after a break in the animation program, the screen looks as though it has new margins and print is oddly formated on the screen. Edit the ANIMATE program so that the address in $58 and $59 will be preserved. Store the starting address of screen RAM in two consecutive memory locations on the zero page. Memory locations $CB-$CF are free bytes of memory. Whenever the pinwheel is moved, update your own screen address rather than interfering with the address stored at $58 and $59.

3. Instead of leaving a zero wherever the pinwheel has been displayed, save what was stored in the screen memory location before putting the pinwheel there. Save the original contents of the memory location on the stack. Then DRAW the pinwheel, ERASE it, and recover the original contents of memory to its former location. For example, if there is an A displayed on the screen, and the pinwheel is about to move into the A's position, push the A onto the stack, and display the pinwheel. Then pull the A off the stack and store it back in its original screen memory location. This way the pinwheel will not erase everything in its path. Instead the screen display will be left in tact.

4. Add some comparisons to the direction subroutines that stop the pinwheel at the edge of the screen. Do not let it wrap around or write over memory above or below screen RAM.

5. Read the joystick for diagonal joystick presses. Incorporate the necessary routines to move the pinwheel on a diagonal as well as up, down, left, and right.

6. Create a shape which is three or four characters wide and high, using keyboard control characters. Animate the shape using input from the joystick. Offer the user a way to exit

the program without pressing <SYSTEM RESET>. Perhaps you could instruct the user to press one of the function keys, such as <SELECT>, to exit the program. In your program you would check for a <SELECT> keypress in the program loop.

Disk File Maintanence

The following instructions will be useful to you as you begin writing your own assembly language programs and saving them on disk. These commands are entered at the EDIT prompt of the Atari Assembler Editor.

<u>Load</u> <u>a</u> <u>Source</u> <u>File</u>: ENTER #D:FILENAME

The ENTER command is used to retrieve a source program from a disk.

<u>Save</u> <u>a</u> <u>Source</u> <u>File</u>: LIST #D:FILENAME

This command is used to save the source code of an assembly language program on a disk. File names must start with a letter and have no more than eight characters, consisting of letters and numbers only.

<u>List</u> <u>the</u> <u>Program</u> <u>on</u> <u>the</u> <u>Printer</u>: LIST #P:

The LIST command can be used to list the source code of an assembly language program in memory on the printer.

<u>List</u> <u>the</u> <u>Assembled</u> <u>Program</u> <u>on</u> <u>the</u> <u>Printer</u>: ASM,#P

To send an assembled version of a program to the printer, specify printer output in the ASM command. This will provide a combined listing of the source code and the object code.

<u>Save</u> <u>the</u> <u>Object</u> <u>Code</u>: SAVE #D:FILENAME<start address, end address

To save the object code of a program on a disk, you must specify the starting and ending hexadecimal addresses of the code in memory in the SAVE command. (eg. SAVE #D:FILENAME<0600,0685)

<u>Load</u> <u>the</u> <u>Object</u> <u>Code</u>: LOAD #D:FILENAME

To retrieve an object program from a disk use the LOAD command in the format shown above. The object code will be reloaded into memory where it was stored when it was SAVED.

```
              25 ;          ARROW
              50 ;
0000          0100            *=    $0600       ;ORIGIN OF PROGRAM
0600 A97D     0110            LDA   #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C   0120            STA   $9C40       ;SCREEN RAM LOCATION
0605 60       0130            RTS               ;RETURN FROM SUBROUTINE
```

```
              25 ;              ARW2
              50 ;
0000          0100            *=    $0600       ;ORIGIN OF PROGRAM
0600 A97D     0110            LDA   #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C   0120            STA   $9C40       ;SCREEN RAM LOCATION
0605 A900     0130            LDA   #$00        ;LOAD ACC. WITH SPACE
0607 8D409C   0140            STA   $9C40       ;STORE SPACE OVER ARROW
060A 60       0150            RTS               ;RETURN FROM SUBROUTINE
```

```
              25 ;            SCRADR
              50 ;
0000          0100            *=    $0600       ;ORIGIN OF PROGRAM
9C40          0105 SCREEN =         $9C40       ;ASSIGN SCREEN
0600 A97D     0110            LDA   #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C   0120            STA   SCREEN      ;STORE ACC. ON SCREEN
0605 60       0130            RTS               ;RETURN FROM SUBROUTINE
```

```
              25 ;              HOLDARROW
              50 ;
0000          0100            *=    $600            ;ORIGIN
9C40          0110 SCREEN =         $9C40
0600 A000     0120            LDY   #$00          ;SET COUNTER
0602 A97D     0130            LDA   #$7D          ;CODE FOR ARROW
0604 8D409C   0140            STA   SCREEN        ;DISPLAY
0607 C8       0150 DELAY      INY               ;ADD ONE TO Y,COUNTER
0608 D0FD     0160            BNE   DELAY         ;IF NOT 0, THEN REPEAT DELAY
060A 60       0170            RTS                  ;RETURN
```

```
           10 ;              POINTER
           20 ;
           30 ;A PROGRAM TO DISPLAY TWO DASHES
           40 ;AND A GREATER THAN SIGN IN THE
           50 ;UPPER LEFT CORNER OF THE SCREEN
           60 ;
           70 ;
0000       0100        *=   $0600      ;ORIGINATE AT $600
0600 A90D  0110        LDA  #$0D       ;LOAD ACC. WITH DASH
0602 8D409C 0120       STA  $9C40      ;STORE ON THE SCREEN
0605 8D419C 0130       STA  $9C41      ;NEXT SCREEN LOCATION
0608 A91E  0140        LDA  #$1E       ;LOAD ACC. WITH >
060A 8D429C 0150       STA  $9C42      ;STORE ON THE SCREEN
060D 00    0160        BRK             ;DISCONTINUE PROGRAM
```

```
           70 ;             AT SIGN
           75 ;
           80 ;DISPLAY AN AT SIGN IN THE
           85 ;MIDDLE OF THE GR. 0 SCREEN.
           90 ;
           95 ;
0000       0100        *=   $600       ;ORG AT $600
9E33       0110 SCREEN =    $9E33      ;SCREEN RAM
0600 A920  0120        LDA  #32        ;LOAD @
0602 8D339E 0130       STA  SCREEN     ;STORE ON SCREEN
0605 00    0140        BRK             ;END PROGRAM
```

```
           10 ;            PINWHEEL
           15 ;
           20 ;THIS PROGRAM USES THE .BYTE
           25 ;PSUEDO OPCODE TO STORE DATA
           30 ;IN MEMORY AND INDEXED ADDRESSING
           35 ;TO READ THROUGH THE DATA.
           40 ;THE PURPOSE OF THE PROGRAM IS
           45 ;TO DISPLAY A SPINNING PINWHEEL
           50 ;IN THE UPPER LEFT HAND CORNER
           55 ;OF THE SCREEN.
           60 ;
           65 ;
0000       0100        *=   $600          ;ORIGIN
9C40       0110 SCREEN =    $9C40     ;SCREEN RAM
0600 A200  0120        LDX  #$00        ;SET INDEX TO 0
0602 BD0E06 0130 NEXTCHAR LDA CHAR,X   ;GET NEXT CHAR
0605 8D409C 0140       STA  SCREEN      ;DISPLAY IT
0608 E8    0150        INX             ;ADD ONE TO INDEX
0609 E004  0160        CPX  #$4         ;COMPARE X REG. TO 4
060B D0F5  0170        BNE  NEXTCHAR    ;IF X=4 THEN BRANCH FOR CHAR
060D 60    0180        RTS             ;RETURN
060E 7C    0190 CHAR   .BYTE 124,15,13,60  ;PINWHEEL
060F 0F
0610 0D
0611 3C
```

```
                              DELAY1
             50 ;
0000         0100        *=    $600          ;ORIGIN
9E33         0110 SCREEN =     $9E33         ;SCREEN RAM
0600 A000    0120        LDY   #$00          ;SET COUNTER
0602 A920    0130        LDA   #$20          ;CODE FOR @
0604 8D339E  0140        STA   SCREEN        ;DISPLAY @
0607 C8      0150 DELAY  INY                 ;ADD 1 TO Y
0608 D0FD    0160        BNE   DELAY         ;IF NOT 0, THEN REPEAT DELAY
060A A900    0164        LDA   #00           ;LOAD ACC. WITH 0
060C 8D339E  0168        STA   SCREEN        ;CLEAR SCREEN
060F A900    0170        LDA   #00           ;LOAD ACC. WITH 0
0611 8D339E  0180        STA   SCREEN        ;CLEAR SCREEN
0614 00      0190        BRK                 ;BREAK
```

```
             10 ;            DELAY2
             15 ;
             20 ;THIS PROGRAM PRINTS AN AT SIGN IN
             30 ;THE CENTER OF THE SCREEN.  A CALL
             40 ;TO A DELAY LOOP HOLDS THE AT SIGN
             50 ;ON THE SCREEN.
             60 ;
             70 ;
0000         0100        *=    $600        ;ORG OF OBJECT CODE
9E33         0110 SCREEN =     $9E33       ;SCREEN RAM
0600 A920    0120        LDA   #$20        ;CODE FOR AN @
0602 8D339E  0130        STA   SCREEN      ;DISPLAY ON SCREEN
0605 200E06  0140        JSR   DELAY       ;WAIT ROUTINE
0608 A900    0145        LDA   #00         ;LOAD ACC. WTIH 0
060A 8D339E  0147        STA   SCREEN      ;ERASE @
060D 00      0150        BRK               ;TERMINATE PROGRAM
             0160 ;
             0170 ;
             0180 ;
060E A2A0    0190 DELAY  LDX   #$A0        ;COUNT Y LOOPS
0610 A000    0200 AGAIN  LDY   #$00        ;0-FF COUNTER
0612 C8      0210 WAIT   INY               ;ADD ONE TO Y
0613 D0FD    0220        BNE   WAIT        ;IF NOT 0, REPEAT WAIT
0615 CA      0230        DEX               ;SUB 1 FROM X
0616 D0F8    0240        BNE   AGAIN       ;IF NOT 0, REPEAT AGAIN
0618 60      0250        RTS               ;RETURN
```

```
               10 ;          SPIN
               20 ;
               30 ;THIS PROGRAM USES FOUR LINES
               40 ;TO PRINT A SPINNING PINWHEEL
               50 ;IN THE UPPER LEFT HAND CORNER
               60 ;OF THE SCREEN.  THE PINWHEEL
               70 ;SPINS ONCE.
               80 ;
               90 ;
               0100 ;
               0110 ;
0000           0120         *=   $600            ;ORIGIN
9C40           0130 SCREEN  =    $9C40      ;SCREEN RAM
0600 A200      0140 DRAW    LDX  #$00          ;SET INDEX TO 0
0602 BD1506    0150 NEXTCHAR LDA CHAR,X     ;GET NEXT CHAR
0605 8D409C    0160         STA  SCREEN        ;DISPLAY IT
0608 8A        0170         TXA               ;TRANSFER X TO ACC.
0609 48        0180         PHA               ;PUSH ACC. ONTO STACK
060A 201906    0190         JSR  DELAY         ;CALL DELAY LOOP
060D 68        0200         PLA               ;PULL ACC. OFF STACK
060E AA        0210         TAX               ;TRANSFER ACC. TO X
060F E8        0220         INX               ;ADD ONE TO INDEX
0610 E004      0230         CPX  #$4           ;COMPARE X REG. TO 4
0612 D0EE      0240         BNE  NEXTCHAR      ;IF X=4 THEN BRANCH FOR CHAR
0614 60        0250         RTS               ;RETURN
0615 7C        0260 CHAR    .BYTE 124,15,13,60  ;PINWHEEL
0616 0F
0617 0D
0618 3C
0619 A255      0270 DELAY   LDX  #$55       ;COUNT 0-255, $55 TIMES
061B A000      0280 AGAIN   LDY  #$00       ;SET COUNTER TO 0
061D C8        0290 WAIT    INY             ;INCREMENT Y REG.
061E D0FD      0300         BNE  WAIT           ;IF NOT 0,WAIT
0620 CA        0310         DEX               ;SUBTRACT 1 FROM X
0621 D0F8      0320         BNE  AGAIN         ;IF NOT 0,AGAIN
0623 60        0330         RTS               ;RETURN
```

```
                10 ;              ANIRIGHT
                20 ;
                30 ;  THIS PROGRAM MOVES THE SPINNING
                40 ;  PINWHEEL TO THE RIGHT, BY
                50 ;  CONTINUALLY ADDING ONE TO THE
                60 ;  SCREEN RAM POSITION.
                70 ;
                80 ;
0000            90            *=    $600
0058            0100 LOWSCR =     $58        ;LOW BYTE OF SCREEN RAM
0059            0110 HISCR  =     $59        ;HIGH BYTE OF SCREEN RAM
                0120 ;
                0130 ;  MAIN LOOP
                0140 ;
0600 200C06     0150 BEGIN   JSR  DRAW       ;DRAW THE PINWHEEL
0603 203406     0160         JSR  DELAY      ;HOLD ON THE SCREEN MOMENTARILY
0606 202606     0170         JSR  RIGHT      ;INCREMENT POSITION TO THE RIGHT
0609 4C0006     0180         JMP  BEGIN      ;REPEAT MAIN LOOP
                0190 ;
                0200 ;
                0210 ;  DRAW READS CHAR DATA AND
                0220 ;  PLACES LINES ON SCREEN IN
                0230 ;  SEQUENCE TO APPEAR LIKE
                0240 ;  SPINNING PINWHEEL.
                0250 ;
                0260 ;
060C A200       0270 DRAW    LDX  #$00       ;SET INDEX TO 0
060E A000       0280         LDY  #$00       ;SET INDEX TO 0
0610 BD2206     0290 NEXTCHAR LDA CHAR,X     ;INDEXED ADDRESSING, GET DATA
0613 9158       0300         STA  (LOWSCR),Y  ;INDIRECT INDEXED ADDRESSING TO SCREEN
0615 8A         0310         TXA             ;TRANSFER X REG. TO ACC.
0616 48         0320         PHA             ;PUSH ACC. ONTO STACK
0617 203406     0330         JSR  DELAY      ;CALL THE DELAY ROUTINE
061A 68         0340         PLA             ;PULL ACC OFF STACK
061B AA         0350         TAX             ;TRANSFER ACC. TO X REG.
061C E8         0360         INX             ;INCREMENT X REGISTER
061D E004       0370         CPX  #$4        ;4 LINES IN PINWHEEL
061F D0EF       0380         BNE  NEXTCHAR   ;GET NEXT CHAR
0621 60         0390         RTS             ;RETURN
0622 7C         0400 CHAR    .BYTE 124,15,13,60 ;PINWHEEL
0623 0F
0624 0D
0625 3C
                0410 ;
                0420 ; RIGHT ADDS ONE TO THE SCREEN
                0430 ; ADDRESS OF THE PINWHEEL
                0440 ;
                0450 ;
0626 18         0460 RIGHT   CLC             ;CLEAR THE CARRY BIT
0627 A558       0470         LDA  LOWSCR        ;GET LOW BYTE OF SCREEN RAM
0629 6901       0480         ADC  #$1           ;ADD 1 AND CARRY TO ACC.
```

```
062B 8558     0490          STA  LOWSCR        ;UPDATE LOWSCR
062D A559     0500          LDA  HISCR         ;GET HIGH BYTE OF SCREEN RAM
062F 6900     0510          ADC  #$00          ;ADD 0 AND CARRY
0631 8559     0520          STA  HISCR         ;UPDATE HIGH BYTE SCREEN RAM
0633 60       0530          RTS              ;RETURN
              0540 ;
              0550 ;
              0560 ;  DELAY HOLDS THE IMAGE
              0570 ;  IN ONE PLACE, MOMENTARILY
              0580 ;  BEFORE THE NEXT MOVE.
              0590 ;
              0600 ;
0634 A219     0610 DELAY  LDX  #$19        ;COUNT 0-255, 25 TIMES
0636 A000     0620 AGAIN  LDY  #$00        ;SET COUNTER TO 0
0638 C8       0630 WAIT   INY              ;ADD 1 TO Y REG.
0639 D0FD     0640          BNE  WAIT          ;IF NOT 0, WAIT
063B CA       0650          DEX              ;SUBTRACT 1 FROM X REG.
063C D0F8     0660          BNE  AGAIN         ;$19 YET?
063E 60       0670          RTS              ;RETURN
```

```
                ;                ERASE
                ;
                ;  THIS PROGRAM MOVES THE SPINNING
                ;  PINWHEEL TO THE RIGHT, BY
                ;  CONTINUALLY INCREMENTING THE
                ;  SCREEN RAM POSITION.  EACH TIME
                ;  THE PINWHEEL IS DRAWN, A SPACE
                ;  IS PRINTED OVER THE LAST PINWHEEL
                ;  POSITION SO NOT TO LEAVE A TRAIL
                ;
                ;
0000                    *=    $600
0058            LOWSCR =     $58       ;LOW BYTE OF SCREEN
0059            HISCR  =     $59       ;HIGH BYTE OF SCREEN RAM
                ;
                ;  MAIN LOOP
                ;
0600 200F06     BEGIN   JSR   DRAW      ;DRAW THE PINWHEEL
0603 203E06             JSR   DELAY     ;HOLD ON THE SCREEN MOMENTARILY
0606 203706             JSR   ERASE     ;ERASE LINE WITH SPACE
0609 202906             JSR   RIGHT     ;INCREMENT POSITION TO THE RIGHT
060C 4C0006             JMP   BEGIN     ;REPEAT MAIN LOOP
                ;
                ;
                ;  DRAW READS CHAR DATA AND
                ;  PLACES LINES ON SCREEN IN
                ;  SEQUENCE TO APPEAR LIKE
                ;  SPINNING PINWHEEL.
                ;
                ;
060F A200       DRAW    LDX   #$00      ;SET INDEX TO 0
0611 A000               LDY   #$00      ;SET INDEX TO 0
0613 BD2506     NEXTCHAR LDA CHAR,X     ;INDEXED ADDRESSING, GET DATA
0616 9158               STA   (LOWSCR),Y  ;INDIRECT INDEXED ADDRESSING TO SCREEN
0618 8A                 TXA             ;TRANSFER X REG. TO ACC.
0619 48                 PHA             ;PUSH ACC. ONTO STACK
061A 203E06             JSR   DELAY     ;CALL THE DELAY ROUTINE
061D 68                 PLA             ;PULL ACC OFF STACK
061E AA                 TAX             ;TRANSFER ACC. TO X REG.
061F E8                 INX             ;INCREMENT X REGISTER
0620 E004               CPX   #$4       ;4 LINES IN PINWHEEL
0622 D0EF               BNE   NEXTCHAR  ;GET NEXT CHAR
0624 60                 RTS             ;RETURN
0625 7C         CHAR    .BYTE 124,15,13,60 ;PINWHEEL
0626 0F
0627 0D
0628 3C
                ;
                ; RIGHT ADDS ONE TO THE SCREEN
                ; ADDRESS OF THE PINWHEEL
                ;
```

```
          0490 ;
0629 A558 0500 RIGHT  LDA  LOWSCR     ;GET LOW BYTE OF SCREEN RAM
062B 18   0510        CLC               ;CLEAR THE CARRY
062C 6901 0520        ADC  #$1           ;ADD 1 AND CARRY TO ACC.
062E 8558 0530        STA  LOWSCR        ;UPDATE LOWSCR
0630 A559 0540        LDA  HISCR         ;GET HIGH BYTE OF SCREEN RAM
0632 6900 0550        ADC  #$00          ;ADD 0 AND CARRY
0634 8559 0560        STA  HISCR         ;UPDATE HIGH BYTE SCREEN RAM
0636 60   0570        RTS               ;RETURN
          0580 ;
          0590 ;
          0600 ;  ERASE PUTS A SPACE OVER THE
          0610 ;  SPINNING PINWHEEL'S LAST POSITION.
          0620 ;
          0630 ;
0637 A000 0640 ERASE  LDY  #$00       ;INDEX
0639 A900 0650        LDA  #$00          ;VALUE FOR SPACE
063B 9158 0660        STA  (LOWSCR),Y    ;STORE IN LAST LOCATION
063D 60   0670        RTS               ;RETURN
          0680 ;
          0690 ;
          0700 ;  DELAY HOLDS THE IMAGE
          0710 ;  IN ONE PLACE, MOMENTARILY
          0720 ;  BEFORE THE NEXT MOVE.
          0730 ;
          0740 ;
063E A225 0750 DELAY  LDX  #$25       ;COUNT 0-255, $25 TIMES
0640 A000 0760 AGAIN  LDY  #$00       ;SET COUNTER TO 0
0642 C8   0770 WAIT   INY             ;ADD 1 TO Y REG.
0643 D0FD 0780        BNE  WAIT          ;IF NOT 0, WAIT
0645 CA   0790        DEX               ;SUBTRACT 1 FROM X REG.
0646 D0F8 0800        BNE  AGAIN         ;$19 YET?
0648 60   0810        RTS               ;RETURN
```

```
                   ;                  JOYMOVE
                   ;
                   ;  THIS PROGRAM MOVES A SPINNING PINWHEEL TO THE LEFT
                   ;  OR THE RIGHT ON THE SCREEN.  THE PINWHEEL'S
                   ;  DIRECTION OF TRAVEL IS CONTROLED
                   ;  BY THE JOYSTICK IN PORT #1.
                   ;
                   ;
0000                         *=    $600
                   ;
0278               STICK  =        $278       ;FEEDBACK FROM JOYSTICK #1
0058               LOWSCR =        $58        ;LOW BYTE OF SCREEN RAM
0059               HISCR  =        $59        ;HIGH BYTE OF SCREEN RAM
                   ;
                   ; MAIN LOOP
                   ;
0600 200F06        BEGIN   JSR   JOYSTICK         ;READ JOYSTICK SUBROUTINE
0603 203706                JSR   DRAW             ;DRAW THE PINWHEEL
0606 205806                JSR   DELAY            ;LEAVE ON THE SCREEN MOMENTARILY
0609 205106                JSR   ERASE            ;ERASE WITH A SPACE
060C 4C0006                JMP   BEGIN            ;JUMP TO BEGIN, REPEAT MAIN LOOP
                   ;
                   ;  READ AND INTERPRET THE VALUE RETURNED FROM THE JOYSTICK
                   ;
060F AD7802        JOYSTICK LDA STICK         ;LOAD ACC WITH CONTENTS OF $278
0612 C907                  CMP   #$7              ;WAS IT PRESSED TO THE RIGHT?
0614 F005                  BEQ   RIGHT            ;IF YES BRANCH TO RIGHT ROUTINE
0616 C90B                  CMP   #$B              ;TO THE LEFT?
0618 F00F                  BEQ   LEFT             ;IF SO BRANCH TO LEFT ROUTINE
061A 60                    RTS
061B 18            RIGHT   CLC                  ;CLEAR THE CARRY BIT
061C A558                  LDA   LOWSCR             ;GET LOW BYTE OF SCREEN RAM
061E 6901                  ADC   #$1              ;ADD 1 AND CARRY TO ACC.
0620 8558                  STA   LOWSCR           ;UPDATE LOWSCR
0622 A559                  LDA   HISCR            ;GET HIGH BYTE
0624 6900                  ADC   #$00             ;ADD CARRY AND ZERO TO HIGH BYTE
0626 8559                  STA   HISCR
0628 60                    RTS
0629 38            LEFT    SEC                  ;SET THE CARRY BIT
062A A558                  LDA   LOWSCR             ;GET LOW BYTE OF SCREEN RAM
062C E901                  SBC   #$1              ;SUBTRACT 1 AND CARRY
062E 8558                  STA   LOWSCR
0630 A559                  LDA   HISCR            ;GET HIGH BYTE SCREEN RAM
0632 E900                  SBC   #$00             ;ANYTHING IN CARRY TO SUBTRACT?
0634 8559                  STA   HISCR            ;UPDATE HIGH BYTE SCREEN RAM
0636 60                    RTS
                   ;
                   ;  DRAW READS CHAR DATA AND PLACES LINES
                   ;  ON SCREEN IN ORDER OF SEQUENCE TO APPEAR LIKE
                   ;  A SPINNING PINWHEEL
                   ;
```

```
063A A200    0520 DRAW    LDX  #$00             ;SET INDEX TO 0
063C A000    0530         LDY  #$00             ;INDEX
063E BD5006  0540 NEXTCHR LDA CHAR,X            ;INDEXED ADDRESSING
0641 9158    0550         STA  (LOWSCR),Y       ;INDEXED INDIRECT ADDRESSING
0643 8A      0560         TXA                   ;TRANSFER X TO ACC.
0644 48      0570         PHA                   ;PUSH ACC. ONTO STACK
0645 205B06  0580         JSR  DELAY            ;JUMP TO DELAY ROUTINE
0648 68      0590         PLA                   ;PULL ACC. OFF STACK
0649 AA      0600         TAX                   ;TRANSFER ACC. TO X REG.
064A E8      0610         INX                   ;INCREMENT X
064B E004    0620         CPX  #$4              ;4 LINES IN PINWHEEL
064D D0EF    0630         BNE  NEXTCHR          ;GET NEXT ONE
064F 60      0640         RTS
0650 7C      0650 CHAR    .BYTE 124,15,13,60 ;VALUES FOR LINES
0651 0F
0652 0D
0653 3C
             0660 ;
             0670 ;   ERASE PUTS A SPACE OVER THE SPINNING
             0680 ;   PINWHEELS LAST POSITION
             0690 ;
0654 A000    0700 ERASE   LDY  #$00             ;INDEX FOR ZERO PAGE ADDRESSING
0656 A900    0710         LDA  #$00             ;VALUE FOR SPACE
0658 9158    0720         STA  (LOWSCR),Y       ;STORE IN LAST LOCATION
065A 60      0730         RTS
             0740 ;
             0750 ;   DELAY HOLDS THE IMAGE IN ONE PLACE MOMENTARILY
             0760 ;   BEFORE READING NEXT MOVE
             0770 ;
065B A219    0780 DELAY   LDX  #$19        ;COUNT 0-255 25 TIMES
065D A000    0790 AGAIN   LDY  #$00
065F C8      0800 WAIT    INY              ;INCREMENT Y REGISTER
0660 D0FD    0810         BNE  WAIT             ;IF NOT ZERO, WAIT
0662 CA      0820         DEX                   ;25 YET?
0663 D0F8    0830         BNE  AGAIN            ;IF NOT ZERO, AGAIN
0665 60      0840         RTS
```

```
            ;                  ANIMATE
            ;
            ;THIS PROGRAM MOVES A SPINNING PINWHEEL AROUND ON THE
            ;GRAPHICS ZERO SCREEN.  THE PINWHEEL IS CONTROLLED BY A
            ;JOYSTICK PLUGGED INTO PORT #1
            ;
            ;
            ;
            ;
0000                 *=    $600
            ;
0278        STICK  =     $278       ;FEEDBACK FROM JOYSTICK #1
0058        LOWSCR =     $58        ;LOW BYTE OF SCREEN RAM
0059        HISCR  =     $59        ;HIGH BYTE OF SCREEN RAM
            ;
            ; MAIN LOOP
            ;
0600 200F06 BEGIN    JSR   JOYSTICK        ;READ JOYSTICK SUBROUTINE
0603 205B06          JSR   DRAW            ;DRAW THE PINWHEEL
0606 207C06          JSR   DELAY           ;LEAVE ON THE SCREEN MOMENTARILY
0609 207506          JSR   ERASE           ;ERASE WITH A SPACE
060C 4C0006          JMP   BEGIN           ;JUMP TO BEGIN, REPEAT MAIN LOOP
            ;
            ;  READ AND INTERPRET THE VALUE RETURNED FROM THE JOYSTICK
            ;
060F AD7802 JOYSTICK LDA STICK       ;LOAD ACC WITH CONTENTS OF $278
0612 C907            CMP   #$7             ;WAS IT PRESSED TO THE RIGHT?
0614 F00D            BEQ   RIGHT           ;IF YES BRANCH TO RIGHT ROUTINE
0616 C90B            CMP   #$B             ;TO THE LEFT?
0618 F017            BEQ   LEFT            ;IF SO BRANCH TO LEFT ROUTINE
061A C90E            CMP   #$E             ;14 FOR UP?
061C F021            BEQ   UP
061E C90D            CMP   #$D             ;13 FOR DOWN?
0620 F02B            BEQ   DOWN
0622 60              RTS
0623 18     RIGHT    CLC                   ;CLEAR THE CARRY BIT
0624 A558            LDA   LOWSCR          ;GET LOW BYTE OF SCREEN RAM
0626 6901            ADC   #$1             ;ADD 1 AND CARRY TO ACC.
0628 8558            STA   LOWSCR          ;UPDATE LOWSCR
062A A559            LDA   HISCR           ;GET HIGH BYTE
062C 6900            ADC   #$00            ;ADD CARRY AND ZERO TO HIGH BYTE
062E 8559            STA   HISCR
0630 60              RTS
0631 38     LEFT     SEC                   ;SET THE CARRY BIT
0632 A558            LDA   LOWSCR          ;GET LOW BYTE OF SCREEN RAM
0634 E901            SBC   #$1             ;SUBTRACT 1 AND CARRY
0636 8558            STA   LOWSCR
0638 A559            LDA   HISCR           ;GET HIGH BYTE SCREEN RAM
063A E900            SBC   #$00            ;ANYTHING IN CARRY TO SUBTRACT?
06[illegible] 8559   STA   HISCR           ;UPDATE HIGH BYTE SCREEN RAM
063E 60              RTS
```

```
063F 38     0520 UP      SEC                  ;SET THE CARRY BIT
0640 A558   0530         LDA  LOWSCR           ;LOAD ACC WITH LOW BYTE
0642 E928   0540         SBC  #$28             ;SUBTACT 40 FROM ACCUMULATOR
0644 8558   0550         STA  LOWSCR
0646 A559   0560         LDA  HISCR
0648 E900   0570         SBC  #$00             ;SUBTRACT ZERO AND CARRY
064A 8559   0580         STA  HISCR
064C 60     0590         RTS
064D 18     0600 DOWN    CLC                    ;CLEAR THE CARRY BIT
064E A558   0610         LDA  LOWSCR            ;GET LOW BYTE OF SCREEN RAM
0650 6928   0620         ADC  #$28              ;ADD 40 ($28) FOR EACH LINE DOWN
0652 8558   0630         STA  LOWSCR
0654 A559   0640         LDA  HISCR             ;GET HIGH BYTE SCREEN RAM
0656 6900   0650         ADC  #$00              ;ADD ANY CARRY
0658 8559   0660         STA  HISCR             ;UPDATE HIGH BTYE
065A 60     0670         RTS
            0680 ;
            0690 ;  DRAW READS CHAR DATA AND PLACES LINES
            0700 ;  ON SCREEN IN ORDER OF SEQUENCE TO APPEAR LIKE
            0710 ;  A SPINNING PINWHEEL
            0720 ;
065B A200   0730 DRAW    LDX  #$00              ;SET INDEX TO 0
065D A000   0740         LDY  #$00              ;INDEX
065F BD7106 0750 NEXTCHR LDA  CHAR,X            ;INDEXED ADDRESSING
0662 9158   0760         STA  (LOWSCR),Y        ;INDEXED INDIRECT ADDRESSING
0664 8A     0770         TXA                    ;TRANSFER X TO ACC.
0665 48     0780         PHA                    ;PUSH ACC. ONTO STACK
0666 207C06 0790         JSR  DELAY             ;JUMP TO DELAY ROUTINE
0669 68     0800         PLA                    ;PULL ACC. OFF STACK
066A AA     0810         TAX                    ;TRANSFER ACC. TO X REG.
066B E8     0820         INX                    ;INCREMENT X
066C E004   0830         CPX  #$4               ;4 LINES IN PINWHEEL
066E D0EF   0840         BNE  NEXTCHR           ;GET NEXT ONE
0670 60     0850         RTS
0671 7C     0860 CHAR    .BYTE 124,15,13,60 ;VALUES FOR LINES
0672 0F
0673 0D
0674 3C
            0870 ;
            0880 ;  ERASE PUTS A SPACE OVER THE SPINNING
            0890 ;  PINWHEELS LAST POSITION
            0900 ;
0675 A000   0910 ERASE   LDY  #$00              ;INDEX FOR ZERO PAGE ADDRESSING
0677 A900   0920         LDA  #$00              ;VALUE FOR SPACE
0679 9158   0930         STA  (LOWSCR),Y        ;STORE IN LAST LOCATION
067B 60     0940         RTS
            0950 ;
            0960 ;  DELAY HOLDS THE IMAGE IN ONE PLACE MOMENTARILY
            0970 ;  BEFORE READING NEXT MOVE
            0980 ;
067C A219   0990 DELAY   LDX  #$19          ;COUNT 0-255 25 TIMES
067E A000   1000 AGAIN   LDY  #$00

0680 C8     1010 WAIT    INY                ;INCREMENT Y REGISTER
0681 D0FD   1020         BNE  WAIT              ;IF NOT ZERO, WAIT
0683 CA     1030         DEX                    ;25 YET?
0684 D0F8   1040         BNE  AGAIN             ;IF NOT ZERO, AGAIN
0686 60     1050         RTS
```

```
10 REM *                    FILLSCREEN
20 REM *
30 REM *  A PROGRAM WHICH FILLS THE SCREEN WITH ONE LETTER
40 REM *  ACCORDING TO THE MOST RECENT KEYPRESS.  AN ASSEMBLY
50 REM *  LANGUAGE ROUTINE IS POKED INTO MEMORY STARTING AT
60 REM *  1536 ($600) USING THE DECIMAL VALUES FOR THE MACHINE
70 REM *  CODE LISTED IN DATA LINES 220-250.  THE PURPOSE
80 REM *  OF THIS PROGRAM IS TO DEMONSTRATE THE SPEED OF AN
90 REM *  ASSEMBLY LANGUAGE ROUTINE.
95 REM **********************************************************
100 REM *
110 REM *  LINES 140-180 READ THE ASSEMBLY ROUTINE
120 REM *  DATA AND POKE IT INTO MEMORY
130 REM *
140 PROGRAMLEN=74:REM ASSEMBLY ROUTINE IS 75 BYTES LONG (0-74)
150 FOR CODE=0 TO PROGRAMLEN
160 READ INSTRUCTION
170 POKE 1536+CODE,INSTRUCTION
180 NEXT CODE
190 REM *
200 REM *  ASSEMBLY ROUTINE DATA
210 REM *
220 DATA 104,104,104,141,77,6,201,0,240,23,201,32,48,4,201,95,48,9,24,105
230 DATA 64,141,77,6,76,33,6,56,233,32,141,77,6,165,88,133,203,165,89,133
240 DATA 204,169,3,141,76,6,169,152,141,75,6,173,77,6,160,0,145,203,230
250 DATA 203,208,2,230,204,206,75,6,208,243,206,76,6,16,238,96
260 PRINT "PRESS ANY KEY";
270 OPEN #2,4,0,"K:"
280 GET #2,CHARACTER
290 REM *  CALL EXECUTES THE ASSEMBLY ROUTINE IN MEMORY
300 CALL=USR(1536,CHARACTER)
310 GOTO 280
```

INTERNAL CHARACTER SET

Column 1				Column 2				Column 3				Column 4			
#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR
0	Space	16	0	32	@	48	P	64	♥	80	♣	96	◆	112	p
1	!	17	1	33	A	49	Q	65	┣	81	┏	97	a	113	q
2	"	18	2	34	B	50	R	66	┃	82	━	98	b	114	r
3	#	19	3	35	C	51	S	67	┛	83	╋	99	c	115	s
4	$	20	4	36	D	52	T	68	┫	84	●	100	d	116	t
5	%	21	5	37	E	53	U	69	┓	85	▄	101	e	117	u
6	&	22	6	38	F	54	V	70	╱	86	▎	102	f	118	v
7	'	23	7	39	G	55	W	71	╲	87	┳	103	g	119	w
8	(	24	8	40	H	56	X	72	◢	88	┻	104	h	120	x
9	)	25	9	41	I	57	Y	73	▗	89	▌	105	i	121	y
10	*	26	:	42	J	58	Z	74	◣	90	┗	106	j	122	z
11	+	27	;	43	K	59	[	75	▝	91[1]	␛	107	k	123	♠
12	,	28	<	44	L	60	\	76	▘	92	↑	108	l	124	|
13	-	29	=	45	M	61	]	77	▔	93	↓	109	m	125[1]	↰
14	.	30	>	46	N	62	^	78	▁	94	←	110	n	126[1]	◀
15	/	31	?	47	O	63	_	79	▖	95	→	111	o	127[1]	▶

1. In mode 0 these characters must be preceded with an escape, CHR$(27), to be printed.

INSTRUCTION SET (OPERATION CODES)

TYPE OF INSTRUCTION	MNEMONIC	OPERATION	TYPE OF ADDRESSING												CONDITION FLAGS (Affected - ●)							
			NON-INDEXED				INDEXED						MISCELLANEOUS Register (A,X,Y,P, PC or SP) Flag (F) Relative (R) Stack (S) None (N)									
			INDIRECT	DIRECT			DIRECT				INDIRECT											
			Indirect	Immed	Page 0	Absolute	Abs ,X	Abs. Y	Indexed, X Page 0	Indexed, Y Page 0	Indexed Indirect	Indirect Indexed										
		Operand ►	($hhhh) (ABS)	#$hh #BY	$hh BY	$hhhh ABS	$hhhh,X ABS,X	$hhhh,Y BY,Y	$hh,X BY,X	$hh,Y BY,Y	($hh,X) (BY,X)	($hh),Y (BY),Y			7	6	5	4	3	2	1	0
		Bytes ►	3	2	2	3	3	3	2	2	2	2	1	2	N	V		B	D	I	Z	C
LOAD & STORE	LDA	M→A Note 1		A9	A5	AD	BD	B9	B5		A1	B1			●						●	
	STA	A→M			85	8D	9D	99	95		81	91										
	LDX	M→X		A2	A6	AE		BE		B6					●						●	
	STX	X→M			86	8E				96												
	LDY	A→Y		A0	A4	AC	BC		B4						●						●	
	STY	Y→M			84	8C			94													
	Machine cycles			②	③	④	④	④	④	④	⑥	⑤										
ARITHMETIC & LOGICAL	AND	A∧M→A Note 1		29	25	2D	3D	39	35		21	31			●						●	
	BIT	A∧M			24	2C									M7	M6					●	
	CMP	A-M		C9	C5	CD	DD	D9	D5		C1	D1			●						●	4
	CPX	X-M		E0	E4	EC									●						●	4
	CPY	Y-M		C0	C4	CC									●						●	4
	ADC	A+M+C→A Note 1,3		69	65	6D	7D	79	75		61	71			●	●					●	●
	SBC	A-M-C→A Note 1,3		E9	E5	ED	FD	F9	F5		E1	F1			●	●					●	●
	ORA	A∨M→A		09	05	0D	1D	19	15		01	11			●						●	
	EOR	A∀M→A		49	45	4D	5D	59	55		41	51			●						●	
	INC	M+1→M			E6⑤	EE⑥	FE⑦		F6⑥						●						●	
	DEC	M-1→M			C6⑤	CE⑥	DE⑦		D6⑥						●						●	
	INX	X+1→X											E8(X)		●						●	
	DEX	X-1→X											CA(X)		●						●	
	INY	Y+1→Y											C8(Y)		●						●	
	DEY	Y-1→Y											88(Y)		●						●	
	Machine cycles			②	③	④	④	④	④		⑥	⑤	②									
SHIFT & ROTATE	ASL	C←[7 0]←0			06	0E	1E		16				0A(A)		●						●	●
	ROL	←[7 0]←[C]←			26	2E	3E		36				2A(A)		●						●	●
	LSR	0→[7 0]→C			46	4E	5E		56				4A(A)		0						●	●
	ROR	→[C]→[7 0]→			66	6E	7E		76				6A(A)		●						●	●
	Machine cycles				⑤	⑥	⑦		⑥				②									
REGISTER TRANSFER	TAX	A→X											AA(A)		●						●	
	TXA	X→A											8A(X)		●						●	
	TAY	A→Y											A8(A)		●						●	
	TYA	Y→A											98(Y)		●						●	
	TSX	SP→X											BA(SP)		●						●	
	TXS	X→SP											9A(X)									
	Machine cycles												②									
SET & CLEAR FLAGS	CLV	0→V											B8(F)			0						
	CLD	0→D											D8(F)						0			
	SED	1→D											F8(F)						1			
	CLI	0→I											58(F)							0		
	SEI	1→I											78(F)							1		
	CLC	0→C											18(F)									0
	SEC	1→C											38(F)									1
	Machine cycles												②									
BRANCH	BPL	Branch if N=0 Note 2												10(R)								
	BMI	Branch if N=1 Note 2												30(R)								
	BVC	Branch if V=0 Note 2												50(R)								
	BVS	Branch if V=1 Note 2												70(R)								
	BCC	Branch if C=0 Note 2												90(R)								
	BCS	Branch if C=1 Note 2												B0(R)								
	BNE	Branch if Z=0 Note 2												D0(R)								
	BEQ	Branch if Z=1 Note 2												F0(R)								
	JMP	Jump	6C			4C③																
	JSR	Jump to Subrout				20⑥																
	RTS	Return fr Subrout												60(S)⑥								
	BRK	Break (Interrupt)												00(P,PC)⑦						1		
	RTI	Return fr Interrupt												40(S)⑥	From Stack							
	Machine cycles		⑤											②								
STACK	PHP	P→S;SP - 1→SP												08(P)③								
	PLP	SP + 1→S;S→P												28(S)④	From Stack							
	PHA	A→S; SP - 1→SP												48(A)③								
	PLA	SP + 1→SP;S→A												68(S)④	●						●	
NO. OP	NOP	No operation												EA(N)②								

A	Accumulator, or contents
X, Y, P	Registers X, Y, P, or contents
SP	Stack Pointer, or contents
S	Stack
M	Memory location (effective address), or contents
M7	Bit 7 of M
∧ V, ∀	Logical AND, OR, XOR
P→Q	P is copied to Q; P unchanged

NOTES. At the head of each column, under TYPE OF ADDRESSING, the correct way to write an Operand is given, in hex, where 'h' represents a hex digit, and symbolically, where 'BY' and 'ABS' represent numbers of one and two bytes, respectively. The number at the head of each column is the number of bytes of that type of instruction.

The circled number at the foot of a column is the number of machine cycles for the instructions in that block, exceptions are indicated by the circled numbers after the Op Code.

1 If the page boundary is crossed, the number of machine cycles is one more than shown.

2 If the condition is true and the branch is taken, the number of machine cycles is one more than shown when the branch is to the same page and two more than shown when the branch is to a different page.

3. Effects of ADC and SBC may be confusing if the D Flag is set. Check results carefully.

4 C=0 when A or X or Y < M, C=1 when A or X or Y ≥ M